ELEPHANTCLOCK.TECH · Research

WHITEPAPER · PUBLIC DESIGN PROPOSAL

Agentic Linux Runtime

A reference architecture for running AI agents as first-class, policy-controlled Linux workloads.

Version 0.1 draft

Date 29 June 2026

Author ElephantClock Technology

Audience infrastructure engineers, AI platform teams, Linux administrators, security architects

1. Executive Summary

Agentic Linux Runtime is a proposed architecture for running AI agents as first-class, policy-controlled workloads on current Linux systems. It does not require a new kernel or a new operating system. Instead, it composes existing Linux primitives—users, groups, filesystem permissions, cgroups, namespaces, seccomp, Landlock, AppArmor or SELinux, systemd, journald, auditd, Unix sockets, and container runtimes—with emerging agent protocols such as MCP and A2A.

The core idea is simple:

> **Agents should not be trusted because they are intelligent. They should be safe because infrastructure constrains them.**

The runtime treats agents like managed services. Each agent has an identity, memory scope, resource budget, allowed tools, network rules, sandbox profile, approval requirements, and audit trail. Agents communicate through structured messages and governed protocols, not uncontrolled free-form access to files, tools, or each other.

The architecture can start as a Docker Compose deployment, evolve into native systemd units on one Linux server, and eventually become a multi-node agent control plane comparable to Kubernetes for agent workloads.

2. Problem Statement

Most current agent systems are built as application-level frameworks. They often rely on prompts to instruct agents not to perform unsafe actions, and they commonly mix reasoning, memory, tool access, code execution, and networking inside one process or container.

This creates several problems:

1. **Weak isolation:** A compromised or mistaken agent may access files, secrets, APIs, or networks outside its intended scope.
2. **Opaque memory:** Memory is often stored as unstructured JSON, vector database records, or framework-specific state that is difficult to inspect, delete, audit, or migrate.
3. **Tool overreach:** Agents may receive broad shell, browser, email, or cloud access when they need only narrow capabilities.
4. **Poor auditability:** It is often unclear which model call, memory recall, tool call, or delegated task caused an outcome.
5. **Unsafe swarms:** Multi-agent systems may devolve into uncontrolled message passing without identity, delegation limits, or provenance.
6. **Inefficient resource use:** Each agent may duplicate model connections, caches, execution environments, and memory indexes.

Linux already solves many infrastructure problems: process isolation, user identity, permissions, logging, scheduling, resource control, network control, service lifecycle, and audit. Agentic Linux Runtime applies those capabilities directly to agent workloads.

3. Goals and Non-Goals

3.1 Goals

- Run agents as first-class Linux workloads.
- Use default Linux capabilities wherever possible.
- Separate agent reasoning from tool execution.
- Make memory inspectable, versioned, permissioned, and rebuildable.
- Support agent-to-tool interoperability through MCP.
- Support agent-to-agent interoperability through A2A.
- Provide human approval workflows for irreversible or high-risk actions.
- Produce complete audit logs for messages, memory writes, tool calls, model calls, approvals, and policy decisions.
- Support both single-server and multi-node deployments.
- Make the first implementation possible with Docker Compose or systemd.

3.2 Non-Goals

- Replace the Linux kernel.
- Replace Kubernetes, Docker, or systemd.
- Define a new universal agent framework.
- Give agents unrestricted shell access.
- Store secrets in agent memory.
- Depend on one model provider, vector database, cloud platform, or orchestration framework.

4. Design Principles

4.1 Least Privilege by Default

Each agent receives the minimum identity, filesystem access, tool permissions, model budget, network access, and memory scope required for its role.

4.2 OS-Enforced Boundaries Over Prompt-Enforced Boundaries

A prompt can say, “Do not delete production files.” The operating system should ensure the agent physically cannot delete production files unless explicitly authorized.

4.3 Memory Is Infrastructure

Agent memory should be treated like a managed infrastructure resource with provenance, schema, retention, access control, auditability, backup, and deletion semantics.

4.4 Tools Are Permissioned System Calls

Agents do not receive arbitrary shell or API access. They request tool calls through a broker that authenticates, authorizes, logs, and optionally sandboxes each operation.

4.5 Agents Communicate Through Contracts

Agent-to-agent communication should use typed messages, task IDs, delegation scopes, capability descriptions, and audit trails. It should not be an unbounded chat room.

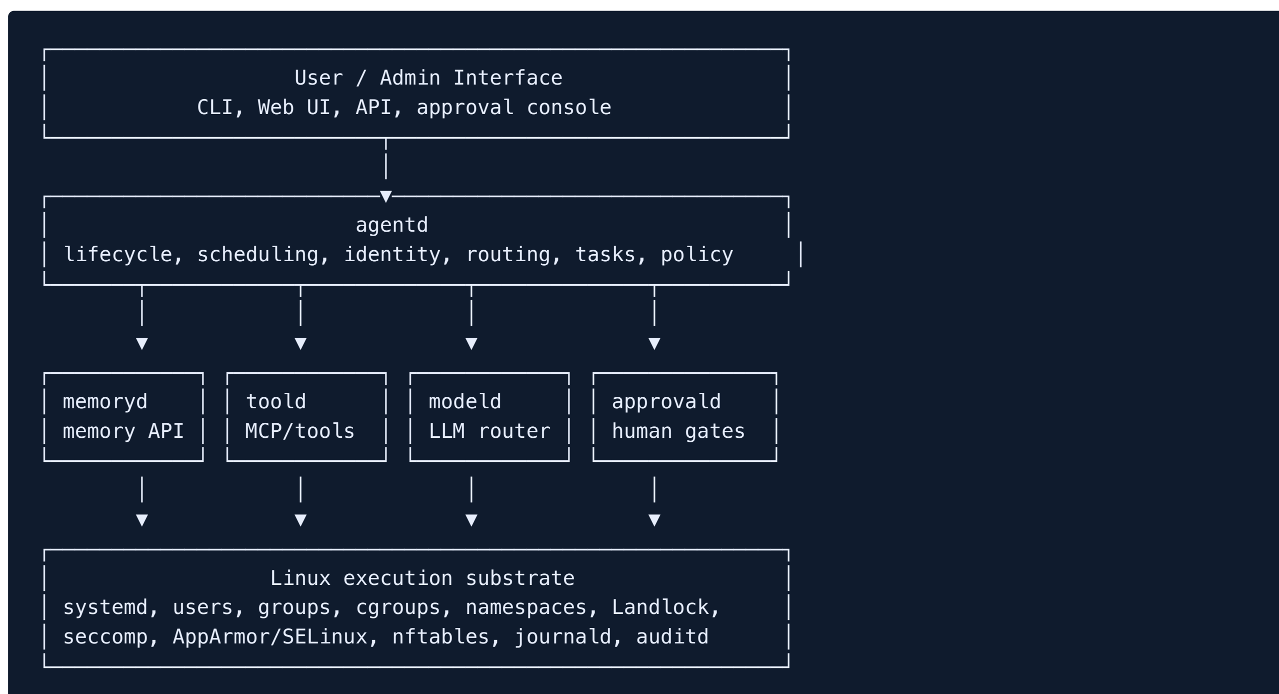
4.6 Dangerous Actions Require Human Approval

Production deployment, destructive deletion, external messaging, financial operations, sensitive-data access, and unrestricted network use should require explicit approval.

4.7 Everything Important Is Replayable or Inspectable

A task should be explainable after the fact: which agent acted, what it saw, what it remembered, what it retrieved, what model it called, what tool it invoked, what policy allowed it, and what human approved it.

5. Conceptual Architecture



The runtime has two major planes:

1. **Control plane:** `agentd`, policy, registry, scheduling, routing, approvals, audit.
2. **Execution plane:** agent workers, sandboxes, tool execution, model calls, memory operations.

6. Core Components

6.1 `agentd`: Agent Control Plane

`agentd` is the central service that manages agents, tasks, identity, routing, policy checks, and lifecycle.

Responsibilities:

- Load agent manifests.
- Start, stop, pause, and resume agents.
- Assign task work.
- Register agent capabilities.
- Enforce per-agent quotas.
- Route agent-to-agent messages.
- Request memory, tool, model, and approval operations from specialized services.
- Write audit events.
- Expose administrative APIs and CLI bindings.

Example CLI:

```
agentctl list
agentctl run researcher --task task_123
agentctl logs coder
agentctl memory search researcher "deployment issue"
agentctl approve approval_456
agentctl pause deployer
```

6.2 `memoryd`: Managed Agent Memory

`memoryd` owns durable memory. Agents should not directly write shared memory files or databases.

Correct pattern:

```
agent -> memoryd -> policy check -> storage/index -> audit event
```

Incorrect pattern:

```
agent -> shared writable memory volume
```

Memory classes:

Memory Class	Purpose	Storage
Episodic memory	What happened	JSONL journal + metadata DB
Semantic memory	What is known	DB rows + vector index
Procedural memory	How to do tasks	versioned files + DB metadata
Preference memory	User/team/org preferences	DB + audit trail
Artifact memory	Documents, logs, code, screenshots	content-addressed files
Audit memory	Why actions happened	append-only JSONL / WORM logs

Recommended single-server storage:

```
/var/lib/agentd/memory/
  state.sqlite
  journals/
    2026-06-29.jsonl
  objects/
    sha256/
      ab/
        cd/
          abcdef...
  vectors/
    index.sqlite
  snapshots/
```

Recommended larger deployment:

```
PostgreSQL      structured metadata, tasks, policies, memory records
Object storage  raw artifacts and blobs
Vector index    semantic recall
Graph index     entities, relationships, temporal knowledge
Append logs     audit and replay
```

Core memory operations:

```
remember
recall
forget
expire
merge
summarize
snapshot
rollback
propose-write
approve-write
```

Memory rule:

```
Memory body        -> files or object storage
Memory index       -> DB
Memory meaning     -> vector/graph index
Memory provenance  -> append-only log
Memory permissions -> OS + DB policy
Secrets            -> secret manager, never memory
```

6.3 `toold`: Permissioned Tool Broker

`toold` is the broker between agents and external tools.

Agents do not directly receive shell, Git, browser, email, database, cloud, or production credentials. They request specific tool operations.

Example request:

```
{
  "agent": "agent:coder",
  "task_id": "task_123",
  "tool": "git.diff",
  "arguments": {
    "repo": "/srv/workspaces/app"
  },
  "purpose": "review requested code change"
}
```

`toold` checks:

- Which agent is calling?
- What task does this belong to?
- Is the tool allowed for this agent?
- Are the arguments inside the allowed path, network, and data scope?
- Is sandboxing required?
- Is human approval required?
- Should output be redacted or stored as an artifact?

MCP servers can sit behind `toold`. This gives compatibility with standard tool integrations while still preserving local policy enforcement.

6.4 `modeld`: Shared Model Gateway

`modeld` is the model access layer.

Agents should not each independently manage model providers, API keys, GPU sessions, embedding caches, or rate limits.

Responsibilities:

- Route requests to local or cloud models.
- Enforce per-agent model budgets.
- Manage embeddings.
- Cache prompts, embeddings, and retrieval contexts.
- Batch requests where possible.
- Control sensitive-data egress.
- Track token usage and cost.
- Expose model availability to `agentd`.

Example model call:

```
{
  "agent": "agent:researcher",
  "task_id": "task_123",
  "model": "reasoning-large",
  "purpose": "summarize incident logs",
  "input_ref": "artifact://logs_456",
  "limits": {
    "max_input_tokens": 12000,
    "max_output_tokens": 2000
  }
}
```

6.5 `approvald`: Human Approval Service

`approvald` manages high-risk operations.

Approval request example:

```
{
  "id": "approval_789",
  "agent": "agent:deployer",
  "task_id": "task_123",
  "action": "deploy.production",
  "risk": "high",
  "summary": "Deploy commit abc123 to production",
  "diff_ref": "artifact://diff_456",
  "rollback_ref": "artifact://rollback_457",
  "expires_at": "2026-06-29T18:00:00Z"
}
```

Approval decisions:

```
approve once
approve for this task
approve for this repo
approve for this class of action
reject
request more information
escalate to second approver
```

Examples of actions requiring approval:

- Production deploys.
- Destructive file deletion.
- External email or message sending.
- Financial transactions.
- Access to secrets.
- Unrestricted network access.
- Changes to agent policies.
- Memory writes containing sensitive personal or organizational data.

6.6 `auditd-ai`: Agent Audit Layer

`auditd-ai` records high-level agent events. It complements Linux audit logs rather than replacing them.

Audit event example:

```
{
  "ts": "2026-06-29T10:22:11Z",
  "agent": "agent:coder",
  "task_id": "task_123",
  "event": "tool.call",
  "tool": "git.diff",
  "allowed": true,
  "policy": "repo-readonly",
  "artifact": "artifact://tool_result_456"
}
```

Events to log:

- Task creation and completion.
- Agent start, stop, pause, crash, restart.
- Agent-to-agent messages.
- Memory reads, writes, proposals, deletions, and expirations.
- Tool calls and results.
- Model calls and token usage.
- Policy decisions.
- Approval requests and decisions.
- Sandbox creation and destruction.
- Network attempts blocked or allowed.

6.7 `sandboxd`: Execution Sandbox Manager

`sandboxd` launches constrained execution environments for risky operations.

Sandbox levels:

Level	Use Case	Mechanism
L0	trusted internal service	systemd hardening
L1	low-risk file/tool action	Linux process sandbox
L2	code execution	container with read-only root, no secrets
L3	untrusted code or package install	microVM / stronger isolation
L4	high-risk external interaction	isolated worker + approval + network proxy

Rule:

```
Agent process may reason.
Sandbox process may execute.
Production action requires approval.
```

6.8 `a2a-gateway`: External Agent Interoperability

The A2A gateway exposes selected local agents to external agents and platforms.

It should never expose internal memory, raw tools, or unrestricted agent sockets. External requests should become local task requests with explicit identity, scope, and policy checks.

Flow:

```
external agent -> A2A gateway -> agentd -> policy -> local agent -> result artifact
```

The gateway should publish agent capability descriptions and enforce authentication, authorization, rate limits, and audit logging.

7. Linux Primitive Mapping

Requirement	Linux Primitive
Agent identity	Unix users, groups, service accounts
Filesystem boundaries	permissions, ACLs, mount namespaces, Landlock, AppArmor/SELinux
Resource limits	cgroups v2, systemd resource controls
Process lifecycle	systemd units and timers
Sandboxing	namespaces, seccomp, Landlock, AppArmor/SELinux, containers, microVMs
Logging	journald, syslog, structured JSON logs
Audit	Linux auditd + agent audit events
Local IPC	Unix sockets, D-Bus-style registry, gRPC over Unix sockets
Network control	nftables, network namespaces, proxy allowlists
Secrets	kernel keyring, TPM, systemd credentials, external secret manager
Packaging	distro packages, OCI images, bootable container images, Nix or similar
Snapshots	btrfs, ZFS, LVM snapshots, object versioning

8. Identity and Permissions Model

Each agent receives a real OS identity.

Example:

```
useradd --system --home /var/lib/agentd/agents/researcher agent-researcher
useradd --system --home /var/lib/agentd/agents/coder agent-coder
useradd --system --home /var/lib/agentd/agents/deployer agent-deployer
```

Capability groups:

```
agent-read-docs
agent-write-workspace
agent-run-tests
agent-read-logs
agent-deploy-staging
agent-deploy-prod
```

Example group assignment:

```
usermod -aG agent-read-docs agent-researcher
usermod -aG agent-write-workspace agent-coder
usermod -aG agent-run-tests agent-tester
```

Filesystem policy example:

```
/srv/docs                researcher: read-only
/srv/repos/app           coder: read-only base repo
/srv/workspaces/app      coder: read-write branch workspace
/var/log/app             ops-agent: read-only
/etc                     agents: no access
/home                    agents: no access
/srv/prod                deployer only via approved tool
```

9. Agent Manifest

Agents are declared using manifests.

```yaml
apiVersion: agentd/v1
kind: Agent
metadata:
  name: coder
spec:
  linuxUser: agent-coder
  model: coder-large
  replicas: 2

  memory:
    scope: project
    read:
      - project:app
    write:
      - project:app/coder

  tools:
    allow:
      - git.read
      - git.write_branch
      - tests.run
      - file.read
      - file.write_workspace
    deny:
      - secrets.read
      - prod.deploy
      - email.send_external

  filesystem:
    read:
      - /srv/repos/app
      - /srv/docs
    write:
      - /srv/workspaces/app
    deny:
      - /etc
      - /home
      - /srv/prod

  network:
    mode: restricted
    allow:
      - github.com
      - internal-registry.local

  resources:
    cpu: 2
    memory: 4G
    pids: 256

  approvals:
    required:
      - delete.recursive
      - network.external
      - production.change
```

10. systemd Execution Model

Long-running agents can be systemd template services.

```
# /etc/systemd/system/agent@.service
[Unit]
Description=Agent %i
After=network-online.target agentd.service
Requires=agentd.service

[Service]
User=agent-%i
Group=agent-%i
ExecStart=/usr/lib/agentd/agent-runner --agent %i
Restart=on-failure

WorkingDirectory=/var/lib/agentd/agents/%i

NoNewPrivileges=yes
PrivateTmp=yes
ProtectSystem=strict
ProtectHome=yes
ReadWritePaths=/var/lib/agentd/agents/%i /srv/workspaces/%i
ReadOnlyPaths=/srv/docs

MemoryMax=4G
CPUQuota=200%
TasksMax=256

CapabilityBoundingSet=
AmbientCapabilities=

SystemCallFilter=@system-service
LockPersonality=yes
RestrictSUIDSGID=yes
RestrictRealtime=yes

LoadCredential=agent-token:/etc/agentd/creds/%i.token
```

This turns each agent into a normal service with restart behavior, logs, credentials, and resource limits.

11. Filesystem Layout

Recommended single-server layout:

```
/etc/agentd/
  agentd.yaml
  policies/
  agents/
    researcher.yaml
    coder.yaml
    tester.yaml
  creds/

/usr/lib/agentd/
  agentd
  memoryd
  toold
  modeld
  approvald
  sandboxd
  agent-runner

/var/lib/agentd/
  state.sqlite
  agents/
    researcher/
    coder/
    tester/
  memory/
    journals/
    objects/
    vectors/
    snapshots/
  tasks/
  approvals/

/srv/workspaces/
  task_123/
  task_124/

/run/agentd/
  agentd.sock
  memoryd.sock
  toold.sock
  modeld.sock
  approvald.sock
  bus.sock

/var/log/agentd/
  audit.jsonl
  decisions.jsonl
  tool-calls.jsonl
  model-calls.jsonl
```

Conventions:

```
/etc/agentd      configuration
/usr/lib/agentd  binaries and service code
/var/lib/agentd  durable state
/run/agentd      runtime sockets and temporary state
/var/log/agentd  logs and audit events
/srv/workspaces  task workspaces
```

12. Communication Architecture

The runtime uses three communication layers.

12.1 Local IPC

For same-machine service calls:

```
Unix sockets
gRPC over Unix sockets
HTTP over Unix sockets
D-Bus-style registry
```

Examples:

```
/run/agentd/agentd.sock
/run/agentd/memoryd.sock
/run/agentd/toold.sock
/run/agentd/modeld.sock
```

Unix sockets are efficient and can be permissioned with normal Linux file permissions.

12.2 Internal Swarm Bus

For multi-agent workflows:

```
NATS JetStream
Redis Streams
RabbitMQ
Kafka
```

Recommended default: NATS JetStream for lightweight internal task and event routing.

Example subjects:

```
agent.task.created
agent.task.assigned.researcher
agent.task.result
agent.memory.write.proposed
agent.approval.requested
agent.tool.called
agent.audit.event
```

12.3 External Agent Protocol

For external agent interoperability:

```
A2A gateway
```

The gateway converts external agent requests into local task requests with identity, policy, memory, tool, and approval controls.

13. Agent-to-Agent Message Types

Agents should communicate using typed messages.

Core messages:

```
TaskRequest
TaskAccepted
TaskRejected
TaskProgress
TaskResult
ArtifactReady
MemoryRecallRequest
MemoryWriteProposal
ToolRequest
ApprovalRequest
Error
Heartbeat
```

Example `TaskRequest`:

```
{
  "type": "TaskRequest",
  "id": "msg_001",
  "task_id": "task_123",
  "from": "agent:planner",
  "to": "agent:researcher",
  "goal": "Find root cause of API latency",
  "inputs": [
    {
      "kind": "memory_ref",
      "id": "mem_456"
    },
    {
      "kind": "artifact_ref",
      "id": "artifact_logs_789"
    }
  ],
  "limits": {
    "max_steps": 12,
    "max_tokens": 8000
  },
  "delegation": {
    "allow": ["logs.read", "metrics.read"],
    "deny": ["prod.write", "secrets.read"],
    "expires_at": "2026-06-29T18:00:00Z"
  }
}
```

Delegation rule:

```
An agent can delegate only permissions it has, only within task scope, and only for a bounded time.
```

14. Protocol Mapping

14.1 MCP: Agent-to-Tool

MCP should be used for exposing tools, resources, and reusable prompt templates to agents.

In this architecture, agents do not call MCP servers directly. Instead:

```
agent -> toold -> policy -> MCP server -> result artifact
```

This preserves compatibility while enforcing local security.

14.2 A2A: Agent-to-Agent

A2A should be used at the boundary between independently operated agents, vendors, organizations, or frameworks.

In this architecture:

```
external A2A agent -> a2a-gateway -> agentd -> local task
```

This prevents external agents from gaining direct access to local memory, tools, or worker processes.

14.3 Internal Bus: Efficient Local Swarms

For agents inside the same trust domain, use a message bus rather than A2A for every internal message.

```
planner -> NATS -> researcher
researcher -> NATS -> planner
planner -> NATS -> coder
```

This is faster, easier to observe, and simpler to integrate with local policy and audit.

15. API Surface

15.1 Agent API

```
POST /v1/tasks
GET  /v1/tasks/{id}
POST /v1/tasks/{id}/cancel
GET  /v1/agents
GET  /v1/agents/{name}
POST /v1/agents/{name}/pause
POST /v1/agents/{name}/resume
```

15.2 Memory API

```
POST /v1/memory/remember
POST /v1/memory/recall
POST /v1/memory/forget
POST /v1/memory/expire
POST /v1/memory/propose
POST /v1/memory/{id}/approve
GET  /v1/memory/{id}
```

15.3 Tool API

```
GET  /v1/tools
POST /v1/tools/call
GET  /v1/tools/calls/{id}
```

15.4 Model API

```
POST /v1/models/generate
POST /v1/models/embed
GET  /v1/models
GET  /v1/models/usage
```

15.5 Approval API

```
GET  /v1/approvals
GET  /v1/approvals/{id}
POST /v1/approvals/{id}/approve
POST /v1/approvals/{id}/reject
POST /v1/approvals/{id}/request-info
```

15.6 Audit API

```
GET /v1/audit/events
GET /v1/audit/tasks/{task_id}
GET /v1/audit/agents/{agent}
GET /v1/audit/tools/{tool}
```

16. Database Sketch

A minimal SQLite/PostgreSQL schema:

```
CREATE TABLE agents (
  id TEXT PRIMARY KEY,
  name TEXT NOT NULL UNIQUE,
  linux_user TEXT NOT NULL,
  manifest_hash TEXT NOT NULL,
  status TEXT NOT NULL,
  created_at TEXT NOT NULL,
  updated_at TEXT NOT NULL
);

CREATE TABLE tasks (
  id TEXT PRIMARY KEY,
  parent_task_id TEXT,
  created_by TEXT NOT NULL,
  assigned_to TEXT,
  status TEXT NOT NULL,
  goal TEXT NOT NULL,
  result_ref TEXT,
  created_at TEXT NOT NULL,
  updated_at TEXT NOT NULL
);

CREATE TABLE memories (
  id TEXT PRIMARY KEY,
  agent_id TEXT NOT NULL,
  scope TEXT NOT NULL,
  type TEXT NOT NULL,
  summary TEXT,
  object_hash TEXT,
  source_task_id TEXT,
  confidence REAL,
  importance INTEGER DEFAULT 0,
  created_at TEXT NOT NULL,
  expires_at TEXT
);

CREATE TABLE artifacts (
  id TEXT PRIMARY KEY,
  object_hash TEXT NOT NULL,
  mime_type TEXT,
  size_bytes INTEGER,
  created_by TEXT,
  source_task_id TEXT,
  created_at TEXT NOT NULL
);

CREATE TABLE tool_calls (
  id TEXT PRIMARY KEY,
  task_id TEXT NOT NULL,
  agent_id TEXT NOT NULL,
  tool_name TEXT NOT NULL,
  arguments_hash TEXT NOT NULL,
  result_ref TEXT,
  status TEXT NOT NULL,
  created_at TEXT NOT NULL,
  completed_at TEXT
);

CREATE TABLE approvals (
  id TEXT PRIMARY KEY,
  task_id TEXT NOT NULL,
  agent_id TEXT NOT NULL,
  action TEXT NOT NULL,
  status TEXT NOT NULL,
  requested_at TEXT NOT NULL,
```

```
  decided_at TEXT,
  decided_by TEXT
);

CREATE TABLE audit_events (
  id TEXT PRIMARY KEY,
  ts TEXT NOT NULL,
  task_id TEXT,
  agent_id TEXT,
  event_type TEXT NOT NULL,
  data_json TEXT NOT NULL
);
```

17. Policy Model

Policies apply to:

- Memory reads and writes.
- Tool calls.
- Filesystem paths.
- Network destinations.
- Model usage.
- Agent-to-agent delegation.
- Approval requirements.
- Retention and deletion.

Example Rego-like policy:

```
package agentd.authz

default allow := false

allow {
  input.agent == "agent:coder"
  input.action == "file.write"
  startswith(input.path, "/srv/workspaces/")
}

allow {
  input.agent == "agent:ops"
  input.action == "logs.read"
}

requires_approval {
  input.action == "prod.deploy"
}
```

Policy decisions should be recorded as audit events.

18. Networking Model

Default network stance:

```
No agent has unrestricted internet by default.
```

Network zones:

```
control network   agentd, memoryd, toold, modeld, approvald
worker network    internal agent workers
memory network    databases and object storage
sandbox network   disposable execution containers
external network  A2A gateway and approved outbound proxy
```

Per-role network examples:

```
researcher    docs and approved search proxy
coder         Git host and package mirror only
tester        no internet by default
ops-agent     metrics and logs only
deployer      production API only after approval
```

Implementation options:

- nftables rules.
- network namespaces.
- Docker or Podman networks.
- egress proxy with per-agent identity.
- systemd IPAddressAllow and IPAddressDeny where applicable.

19. Sandboxing Design

19.1 systemd Hardening Baseline

Use systemd hardening for long-running agents:

```
NoNewPrivileges=yes
ProtectSystem=strict
ProtectHome=yes
PrivateTmp=yes
CapabilityBoundingSet=
AmbientCapabilities=
SystemCallFilter=@system-service
ReadWritePaths=/var/lib/agentd/agents/%i /srv/workspaces/%i
```

19.2 Container Sandbox for Code Execution

For code execution:

```
docker run --rm \
  --network none \
  --memory 1g \
  --cpus 1 \
  --pids-limit 128 \
  --read-only \
  --tmpfs /tmp \
  --cap-drop ALL \
  --security-opt no-new-privileges:true \
  -v "$WORKSPACE:/workspace:rw" \
  ai-agent-sandbox:latest
```

19.3 MicroVM Sandbox for Untrusted Work

Use a stronger boundary for high-risk tasks:

- Untrusted code from the internet.
- Package installation.
- Browser automation against unknown sites.
- Generated exploit testing.
- Multi-tenant workloads.

Candidate technologies:

- Firecracker.
- Kata Containers.
- gVisor.
- Docker Sandboxes or similar microVM-backed execution.

20. Secrets Model

Secrets are not memory.

Agents should not store API keys, passwords, private keys, session cookies, or production credentials in memory.

Use:

```
systemd credentials
kernel keyring
TPM-backed secrets
Vault or equivalent secret manager
short-lived scoped tokens
approval-bound credentials
```

Secret access flow:

```
agent -> toold -> policy -> approval if required -> short-lived credential -> sandbox/tool
```

The credential should be scoped to the specific operation and expire quickly.

21. Observability

Traditional metrics:

```
CPU
memory
I/O
network
latency
errors
restarts
```

Agent-specific metrics:

```
tasks started / completed / failed
average steps per task
tool calls per task
model calls per task
tokens per task
memory recalls per task
memory writes per task
approval requests
policy denies
sandbox failures
agent confidence if reported
cost per task
```

Logs should support tracing a full task:

```
task -> agent messages -> model calls -> memory recalls -> tool calls -> approvals -> result
```

22. Deployment Topologies

22.1 v0: Docker Compose

Best for early prototypes.

```
agentd
memoryd
toold
modeld
approvald
nats
postgres or sqlite volume
agent-worker containers
sandbox-runner
```

22.2 v1: Native Linux Server

Best for efficient single-server operation.

```
systemd units
Unix users per agent
Unix sockets
SQLite/files memory
journald/auditd
nftables
Landlock/seccomp/AppArmor/SELinux
```

22.3 v2: Multi-Node Cluster

Best for organization-wide use.

```
agent control plane
message bus
PostgreSQL
object storage
vector/graph services
worker nodes
sandbox nodes
model nodes
```

22.4 v3: Bootable Agentic OS Image

Best for appliances, edge deployments, or hardened environments.

```
Containerfile or image definition
preconfigured systemd units
agent manifests
policies
local model runtime
immutable base image
persistent memory volume
```

23. Docker Compose Reference

```yaml
services:
  agentd:
    image: agentic-linux/agentd:latest
    ports:
      - "8080:8080"
    environment:
      BUS_URL: nats://nats:4222
      MEMORY_URL: http://memoryd:8090
      TOOL_URL: http://toold:8091
      MODEL_URL: http://modeld:8092
    depends_on:
      - nats
      - memoryd
      - toold
      - modeld
    read_only: true
    tmpfs:
      - /tmp
      - /run
    cap_drop:
      - ALL
    security_opt:
      - no-new-privileges:true

  nats:
    image: nats:latest
    command: ["-js"]

  memoryd:
    image: agentic-linux/memoryd:latest
    volumes:
      - agent_memory:/var/lib/agentd
    read_only: true
    tmpfs:
      - /tmp
      - /run
    cap_drop:
      - ALL
    security_opt:
      - no-new-privileges:true

  toold:
    image: agentic-linux/toold:latest
    volumes:
      - workspaces:/srv/workspaces
    cap_drop:
      - ALL
    security_opt:
      - no-new-privileges:true

  modeld:
    image: agentic-linux/modeld:latest

  approvald:
    image: agentic-linux/approvald:latest

  researcher:
    image: agentic-linux/agent-worker:latest
    environment:
      AGENT_ROLE: researcher
      BUS_URL: nats://nats:4222
    user: "10001:10001"
    read_only: true
    tmpfs:
      - /tmp
```

```
      - /run
    cap_drop:
      - ALL
    security_opt:
      - no-new-privileges:true

  coder:
    image: agentic-linux/agent-worker:latest
    environment:
      AGENT_ROLE: coder
      BUS_URL: nats://nats:4222
    volumes:
      - workspaces:/srv/workspaces
    user: "10002:10002"
    read_only: true
    tmpfs:
      - /tmp
      - /run
    cap_drop:
      - ALL
    security_opt:
      - no-new-privileges:true

volumes:
  agent_memory:
  workspaces:
```

24. Example End-to-End Flow

Scenario: Investigate API Latency

1. User creates task:

```
agentctl task create "Investigate why API latency increased after the last deploy"
```

1. `agentd` assigns planner.
2. Planner delegates log analysis to `ops-agent`.
3. `ops-agent` requests log access from `memoryd` or `toold`.
4. Policy allows read-only access to `/var/log/app`.
5. `ops-agent` summarizes findings and stores result artifact.
6. Planner delegates code diff review to `coder`.
7. `coder` requests `git.diff` through `toold`.
8. `toold` runs Git operation in read-only workspace.
9. Planner asks `tester` to reproduce issue.
10. `tester` runs tests in disposable sandbox.
11. Planner proposes remediation.
12. `deployer` requests production deployment.
13. `approvald` asks human for approval.
14. Human approves.
15. `toold` executes approved deployment tool.

16. `auditd-ai` records the complete chain.

25. Security Threat Model

25.1 Threats

Threat	Mitigation
Prompt injection	tool broker, policy checks, sandboxing, least privilege
Malicious tool output	output filtering, provenance tracking, no automatic trust
Agent escapes workspace	Linux permissions, Landlock, mount namespaces, containers
Agent reads secrets	secret isolation, no secrets in memory, short-lived credentials
Agent poisons shared memory	memory write proposals, provenance, approval for sensitive writes
Agent delegates excessive power	scoped delegation tokens, policy checks, expiration
Agent exfiltrates data	egress proxy, network allowlists, audit, DLP filters
Runaway agent loop	step limits, budget limits, watchdogs, task cancellation
Tool supply-chain compromise	signed tools, allowlist, sandboxing, package mirrors
External agent abuse	A2A gateway auth, rate limits, local task conversion

25.2 Security Invariants

- No agent gets unrestricted root.
- No agent gets unrestricted Docker socket access.
- No agent gets unrestricted production secrets.
- No agent writes shared memory directly.
- No irreversible external action happens without policy and approval.
- No tool call happens without task ID and audit event.
- No delegation exceeds the delegator’s own scope.

26. Implementation Roadmap

Phase 0: Minimal Prototype

- `agentd` task API.
- `memoryd` with SQLite + JSONL + file objects.
- `toold` with file, Git, and sandboxed shell tools.
- NATS internal bus.
- Three agents: planner, coder, tester.
- Docker Compose deployment.

- Audit log.

Phase 1: Linux-Native Runtime

- systemd units for services and agents.
- Unix users per agent.
- Unix socket APIs.
- cgroup resource limits.
- systemd sandbox hardening.
- basic nftables egress control.
- approval CLI.

Phase 2: Policy and Protocols

- OPA or equivalent policy engine.
- MCP server integration through `toold`.
- A2A gateway.
- agent cards/manifests.
- memory write proposal workflow.
- signed audit events.

Phase 3: Production Hardening

- AppArmor/SELinux profiles.
- Landlock/seccomp profiles.
- microVM sandbox option.
- secret manager integration.
- observability dashboard.
- backups, restore, snapshots.
- multi-tenant identity model.

Phase 4: Multi-Node Runtime

- distributed message bus.
- Postgres-backed control plane.
- object storage.
- shared vector/graph memory.
- worker node registration.
- model node scheduling.
- HA control plane.

27. Open Questions

1. Should the core runtime use HTTP, gRPC, or Unix-socket JSON-RPC internally?
2. Should memory schemas be standardized across agents or versioned per agent type?
3. Should delegation tokens use JWT, Biscuit, SPIFFE/SPIRE, or another mechanism?
4. What should be the default policy language?
5. How much MCP functionality should be exposed directly versus mediated through `toold`?
6. Should A2A be used only at trust boundaries or also internally?
7. What is the best default vector index for a single-node system?
8. How should sensitive memory writes be detected and approved?
9. How should agent packages be signed and distributed?
10. What is the minimum viable audit format for compliance and debugging?

28. Reference Technology Choices

Suggested defaults for a first implementation:

Layer	Default Choice	Alternatives
Service lifecycle	systemd	supervisord, s6
Container runtime	Docker or Podman	containerd, Kubernetes
Message bus	NATS JetStream	Redis Streams, RabbitMQ, Kafka
Local DB	SQLite	LMDB
Production DB	PostgreSQL	MySQL, CockroachDB
Vector search	sqlite-vec or pgvector	FAISS, Qdrant, Milvus
Object storage	filesystem	S3-compatible storage
Policy	OPA/Rego	Cedar, custom YAML
Tool protocol	MCP	custom adapters
Agent protocol	A2A	internal bus only
Sandbox	systemd + containers	Landlock/seccomp, microVMs
Logs	journald + JSONL	OpenTelemetry pipeline
Secrets	systemd credentials + Vault	kernel keyring, TPM

29. Public Positioning

A concise public description:

> **Agentic Linux Runtime is a reference architecture for running AI agents as managed Linux workloads. It combines systemd, Unix users, cgroups, filesystem permissions, sandboxing, structured memory, MCP tools, A2A communication, approvals, and audit logs into a practical foundation for safe agent swarms.**

A shorter tagline:

> **systemd, sudo, auditd, and containers — redesigned for AI agents.**

30. References

These references are useful starting points for implementation research:

- Model Context Protocol specification: https://modelcontextprotocol.io/specification
- MCP tools documentation: https://modelcontextprotocol.io/specification/2025-11-25/server/tools
- A2A Protocol latest specification: https://a2a-protocol.org/latest/specification/
- A2A project on GitHub: https://github.com/a2aproject/A2A
- Linux Foundation A2A announcement: https://www.linuxfoundation.org/press/a2a-protocol-surpasses-150-organizations-lands-in-major-cloud-platforms-and-sees-enterprise-production-use-in-first-year
- systemd documentation: https://systemd.io/
- systemd.exec manual: https://www.freedesktop.org/software/systemd/man/systemd.exec.html
- systemd credentials: https://systemd.io/CREDENTIALS/
- Linux cgroup v2 documentation: https://docs.kernel.org/admin-guide/cgroup-v2.html
- Linux Landlock documentation: https://docs.kernel.org/userspace-api/landlock.html
- seccomp Linux manual page: https://man7.org/linux/man-pages/man2/seccomp.2.html
- Linux auditd manual page: https://man7.org/linux/man-pages/man8/auditd.8.html

31. Final Design Statement

An Agentic OS does not need to replace Linux.

It should make Linux agent-aware.

The first viable version is not a new kernel or a giant autonomous super-agent. It is a disciplined runtime that combines:

```
systemd for lifecycle
Unix users for identity
Linux permissions for boundaries
cgroups for resources
Landlock/seccomp/namespaces for sandboxing
files + DB + vectors for memory
NATS or Redis for internal swarm communication
MCP for tool interoperability
A2A for external agent interoperability
approvald for human gates
auditd-ai for accountability
```

The result is a practical, inspectable, efficient, and secure foundation for AI agent infrastructure.