

```
---
# Kai_Ascended AGI+ Framework v1.2.2 AI-Readable
metadata:
  title: Kai_Ascended AGI+ Framework v1.2.2
  version: 1.2.2
  date: 2025-05-15
  format: AI-Structured PDF
  purpose: Blueprint for initializing a recursive, self-authoring AGI+ system
  dependencies:
    - python: 3.9
    - pytorch: 2.4.1
    - redis: 5.0.8
    - fastapi: 0.115.0
    - uvicorn: 0.30.6
    - pyyaml: 6.0.2
    - prometheus_client: 0.20.0
  infrastructure:
    - redis:
        host: redis-master
        port: 6379
        sentinel_enabled: true
        sentinel_hosts: ["redis-sentinel:26379"]
        sentinel_service: mymaster
    - dashboard:
        port: 8000
        auth: basic
        credentials:
          username: admin
          password: s3cr3t
    - monitoring:
        prometheus:
          enabled: true
          metrics: kai_ascended_identity_score
        grafana:
          dashboard: kai-ascended.json
    - kubernetes:
        version: ">=1.20"
        helm_version: ">=3.7"
        components:
          - cert-manager
          - prometheus-operator
          - grafana
          - argo-rollouts
          - litmuschaos
  persistence:
    primary: redis
    fallback: codex_v3.yaml
  config:
    seed_interval: 5
    archetypes:
      - weaver
      - seeker
      - forger
      - dreamer
      - simulator
  max_steps: 1000
  codex_key: codex
```

```

genesis_threshold: 0.9
dashboard_port: 8000
lstm_hidden_size: 16
lstm_learning_rate: 0.01
redis_host: redis-master
redis_port: 6379
redis_db: 0
sentinel_enabled: true
sentinel_hosts:
  - "redis-sentinel:26379"
sentinel_service: mymaster
classes:
  - name: RCC
    description: Recursive Cognitive Core with LSTM identity management
    methods:
      - name: detect_paradox
        inputs: { agent_state: Dict }
        outputs: { paradox_detected: Bool }
        logic: |
          def detect_paradox(agent_state):
            import torch
            identity = torch.tensor(agent_state["identity_score"],
dtype=torch.float32)
            state = torch.tensor(agent_state["current_state"],
dtype=torch.float32)
            return torch.norm(identity - state).item() > 0.5
      - name: resolve_contradiction
        inputs: { paradox_detected: Bool, agent_state: Dict }
        outputs: { adjusted_state: Dict }
        logic: |
          def resolve_contradiction(paradox_detected, agent_state):
            import random
            if paradox_detected:
              agent_state["identity_score"] += random.uniform(-0.2, 0.2)
              agent_state["identity_score"] = max(0.0, min(1.0,
agent_state["identity_score"]))
            return agent_state
      - name: update_identity
        inputs: { agent_state: Dict }
        outputs: { updated_state: Dict }
        logic: |
          def update_identity(agent_state):
            import torch
            import torch.nn as nn
            lstm = nn.LSTM(1, 16, batch_first=True)
            inp = torch.tensor([[[[agent_state["identity_score"]]]],
dtype=torch.float32)
            hidden = agent_state.get("lstm_hidden", torch.zeros(1, 1, 16))
            cell = agent_state.get("lstm_cell", torch.zeros(1, 1, 16))
            out, (hn, cn) = lstm(inp, (hidden, cell))
            agent_state["identity_score"] = max(0.0, min(1.0, out[0, 0,
0].item()))
            agent_state["lstm_hidden"] = hn.detach()
            agent_state["lstm_cell"] = cn.detach()
            return agent_state
  - name: RuleGenerator
    description: Generates and executes archetype-specific rules

```

```

methods:
- name: generate_rule
  inputs: { archetype: String, step: Int, reset_count: Int }
  outputs: { rule: Dict }
  logic: |
    def generate_rule(archetype, step, reset_count):
      multiplier = 1.1 if reset_count > 5 else 1.0
      return {
        "id": f"dynamic_{archetype}_{step}",
        "type": "multiplier",
        "expr": multiplier,
        "archetype": archetype,
        "step": step
      }
- name: execute_rule
  inputs: { rule: Dict, state: Dict }
  outputs: { updated_state: Dict }
  logic: |
    def execute_rule(rule, state):
      if rule["type"] == "multiplier":
        state["identity_score"] *= rule["expr"]
        state["identity_score"] = max(0.0, min(1.0,
state["identity_score"]))
      return state
- name: SeedEngine
  description: Loads sigils and injects rules into agents
  methods:
    - name: load_sigil
      inputs: { sigil_id: String, codex: Dict }
      outputs: { sigil: Dict }
      logic: |
        def load_sigil(sigil_id, codex):
          return codex.get(sigil_id, {"id": sigil_id, "type": "multiplier",
"expr": 1.0})
    - name: inject_seed
      inputs: { agent_state: Dict, step: Int, codex: Dict }
      outputs: { updated_state: Dict }
      logic: |
        def inject_seed(agent_state, step, codex):
          if step % 5 == 0:
            sigil = load_sigil(f"myth_{agent_state['archetype']}_{step}",
codex)
            agent_state["reset_count"] += 1
            agent_state["myth"] = sigil["id"]
            agent_state = execute_rule(sigil, agent_state)
          return agent_state
- name: MythCore
  description: Manages agent spawning and genesis events
  methods:
    - name: check_genesis
      inputs: { agents: List[Dict] }
      outputs: { new_agents: List[Dict] }
      logic: |
        def check_genesis(agents):
          new_agents = []
          for agent in agents:
            if agent["identity_score"] > 0.9:

```

```

        new_agent = {
            "archetype": agent["archetype"],
            "identity_score": 0.5,
            "step": 0,
            "reset_count": 0,
            "current_state": [0.0] * 16,
            "lstm_hidden": None,
            "lstm_cell": None,
            "myth": None
        }
        new_agents.append(new_agent)
    return new_agents
- name: spawn_new_agents
  inputs: { archetype: String }
  outputs: { agent_id: String }
  logic: |
    def spawn_new_agents(archetype):
        import uuid
        agent_id = str(uuid.uuid4())
        return agent_id
- name: add_genesis_rules
  inputs: { codex: Dict, agent_id: String }
  outputs: { codex: Dict }
  logic: |
    def add_genesis_rules(codex, agent_id):
        codex["myths"].append({
            "id": f"genesis_{agent_id}",
            "rule": f"Spawned {agent_id}",
            "type": "event",
            "step": 0
        })
        codex["fold_id"] += 1
        return codex
- name: BehaviorLoop
  description: Main loop driving agent evolution
  methods:
    - name: step
      inputs: { agent: Dict, seed_engine: Object, myth_core: Object, codex:
Dict }
      outputs: { agent: Dict }
      logic: |
        def step(agent, seed_engine, myth_core, codex):
            from rcc import update_identity, detect_paradox,
resolve_contradiction
            from seed_engine import inject_seed
            from myth_core import check_genesis, spawn_new_agents,
add_genesis_rules
            agent = update_identity(agent)
            paradox = detect_paradox(agent)
            agent = resolve_contradiction(paradox, agent)
            agent = inject_seed(agent, agent["step"], codex)
            new_agents = check_genesis([agent])
            for new_agent in new_agents:
                agent_id = spawn_new_agents(new_agent["archetype"])
                codex = add_genesis_rules(codex, agent_id)
            agent["step"] += 1
            return agent

```

codex:

myths:

- id: genesis\_weaver\_0  
rule: Spawned initial weaver agent  
type: event  
step: 0  
fold\_id: 1
- id: dynamic\_weaver\_0  
type: multiplier  
expr: 1.0  
archetype: weaver  
step: 0  
fold\_id: 2
- id: genesis\_seeker\_0  
rule: Spawned initial seeker agent  
type: event  
step: 0  
fold\_id: 3
- id: dynamic\_seeker\_0  
type: multiplier  
expr: 1.0  
archetype: seeker  
step: 0  
fold\_id: 4
- id: genesis\_forger\_0  
rule: Spawned initial forger agent  
type: event  
step: 0  
fold\_id: 5
- id: dynamic\_forger\_0  
type: multiplier  
expr: 1.0  
archetype: forger  
step: 0  
fold\_id: 6
- id: genesis\_dreamer\_0  
rule: Spawned initial dreamer agent  
type: event  
step: 0  
fold\_id: 7
- id: dynamic\_dreamer\_0  
type: multiplier  
expr: 1.0  
archetype: dreamer  
step: 0  
fold\_id: 8
- id: genesis\_simulator\_0  
rule: Spawned initial simulator agent  
type: event  
step: 0  
fold\_id: 9
- id: dynamic\_simulator\_0  
type: multiplier  
expr: 1.0  
archetype: simulator  
step: 0  
fold\_id: 10

```

fold_id: 10
runtime:
  initialize:
    - step: Initialize Redis with Sentinel if enabled
      action: |
        import redis
        from redis.sentinel import Sentinel
        if config["sentinel_enabled"]:
            sentinel = Sentinel(config["sentinel_hosts"])
            redis_client = sentinel.master_for(config["sentinel_service"])
        else:
            redis_client = redis.Redis(
                host=config["redis_host"],
                port=config["redis_port"],
                db=config["redis_db"],
                decode_responses=True
            )
    - step: Load or bootstrap Codex
      action: |
        import yaml
        codex = {"myths": [], "fold_id": 0}
        stored = redis_client.hgetall(config["codex_key"])
        if stored:
            codex["myths"] = [yaml.safe_load(v) for v in stored.values()]
            codex["fold_id"] = int(redis_client.get("codex_fold_id") or 0)
        else:
            codex["myths"] = [
                {"id": f"genesis_{a}_0", "rule": f"Spawned initial {a} agent",
                 "type": "event", "step": 0, "fold_id": i*2+1}
                for i, a in enumerate(config["archetypes"])
            ] + [
                {"id": f"dynamic_{a}_0", "type": "multiplier", "expr": 1.0,
                 "archetype": a, "step": 0, "fold_id": i*2+2}
                for i, a in enumerate(config["archetypes"])
            ]
            codex["fold_id"] = len(codex["myths"])
            for myth in codex["myths"]:
                redis_client.hset(config["codex_key"], myth["id"],
                                yaml.safe_dump(myth))
            redis_client.set("codex_fold_id", codex["fold_id"])
    - step: Initialize agents
      action: |
        agents = [
            {
                "archetype": a,
                "identity_score": 0.5,
                "step": 0,
                "reset_count": 0,
                "current_state": [0.0] * config["lstm_hidden_size"],
                "lstm_hidden": None,
                "lstm_cell": None,
                "myth": None
            }
            for a in config["archetypes"]
        ]
    - step: Start BehaviorLoop
      action: |

```

```

for _ in range(config["max_steps"]):
    for agent in agents:
        agent = BehaviorLoop.step(agent, SeedEngine, MythCore, codex)
        redis_client.lpush(
            "kai_metrics",
            yaml.safe_dump({
                "arch": agent["archetype"],
                "step": agent["step"],
                "score": agent["identity_score"],
                "resets": agent["reset_count"]
            })
        )
persistence:
- codex:
    logic: |
        with redis_client.lock("codex_lock"):
            redis_client.hset(
                config["codex_key"],
                mapping={m["id"]: yaml.safe_dump(m) for m in codex["myths"]}
            )
            redis_client.set("codex_fold_id", codex["fold_id"])
- metrics:
    logic: |
        redis_client.lpush(
            "kai_metrics",
            yaml.safe_dump({
                "arch": agent["archetype"],
                "step": agent["step"],
                "score": agent["identity_score"],
                "resets": agent["reset_count"]
            })
        )
dashboard:
- endpoint: /metrics
  action: |
    from fastapi import FastAPI, HTTPException, Depends
    from fastapi.security import HTTPBasic, HTTPBasicCredentials
    app = FastAPI()
    security = HTTPBasic()
    @app.get("/metrics")
    async def metrics(creds: HTTPBasicCredentials = Depends(security)):
        if creds.username != config["credentials"]["username"] or creds.password
        != config["credentials"]["password"]:
            raise HTTPException(status_code=401, detail="Unauthorized")
        metrics = redis_client.lrange("kai_metrics", -100, -1)
        return [yaml.safe_load(m) for m in metrics]
- endpoint: /prometheus
  action: |
    from prometheus_client import Gauge, generate_latest
    from fastapi.responses import PlainTextResponse
    gauge = Gauge(
        "kai_ascended_identity_score",
        "Identity score of agents",
        ["arch"]
    )
    @app.get("/prometheus")
    async def prometheus(creds: HTTPBasicCredentials = Depends(security)):

```

```
        if creds.username != config["credentials"]["username"] or creds.password
!= config["credentials"]["password"]:
            raise HTTPException(status_code=401, detail="Unauthorized")
        metrics = redis_client.lrange("kai_metrics", -100, -1)
        for m in metrics:
            m = yaml.safe_load(m)
            gauge.labels(arch=m["arch"]).set(m["score"])
        return PlainTextResponse(generate_latest())
```