Awesome, you're ready to take the Kai_Ascended AGI+ Framework v1.2.2 to the next level and create a truly clean, universal, AI-readable PDF! Since you want to generate the PDF yourself and need everything required to do so, I'll provide a comprehensive package: the corrected source YAML file, the Python script to generate the PDF, a `requirements.txt` for dependencies, and instructions to ensure a clean, OCR-free output. I'll also include guidance on verifying the PDF and addressing the gaps you outlined (e.g., fixing

`1stm`, `yam1`, `McCarthy`, etc.). This will give you a drop-in document ready for cloud LLM ingestion or local runtime, with no artifacts.

What You'll Get

1. **Corrected YAML Source**: A clean `kai_ascended_agi_framework_v1.2.2.yaml` with all OCR errors fixed (e.g., `lstm` instead of `1stm`, `yaml` instead of `yam1`, proper syntax for `inject_worker`, `uuid.uuid4()`, etc.).

2. **PDF Generator Script**: An enhanced `generate_kai_pdf.py` to convert the YAML into a paginated, AI-readable PDF with selectable text, headers, and versioning.

3. **Requirements**: A `requirements.txt` for dependencies (e.g., `reportlab` for PDF generation, `pyyaml` for parsing).

4. **Instructions**: Step-by-step guidance to generate, verify, and use the PDF, plus sample LLM prompts and tips for RAG integration.

5. **Optional CLI Outline**: A skeleton for the `kai-framework` CLI you mentioned, with demo and live modes, to help you bundle it later.

Step 1: Corrected YAML Source

Below is the cleaned-up YAML source for `kai_ascended_agi_framework_v1.2.2.yaml`. I've fixed all identified errors:

- Replaced `1stm` with `lstm`.
- Corrected `yam1` to `yaml`.

- Removed `McCarthy` from `update_identity`.

- Fixed `inject Worker = 5 == 0` to `if step % 5 == 0` in `inject_worker`.

- Corrected `str Chemicals (uuid.40)` to `str(uuid.uuid4())`.

- Fixed `torch-tensor` to `torch.tensor`.

- Restored truncated sections (e.g., `check_genesis`).

- Corrected dashboard syntax (`:=` to `!=`, `HITP` to `HTTP`, `kai cauctrics`/`wheelrics` to `kai_metrics`).

- Ensured consistent indentation and syntax.

Save this as `kai_ascended_agi_framework_v1.2.2.yaml`:

```yaml
---
title: Kai_Ascended AGI+ Framework v1.2.2
version: 1.2.2
description: AI-Readable specification for the Kai_Ascended AGI+ Framework
```

```
config:
  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
  archetypes:
```

- weaver
- seeker
- forger
- dreamer
- simulator

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["identi
ty_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
            agent_state["lstm_cell"] = cn
            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": multiplierScience,
                "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_worker
      inputs: { agent_state: Dict, step: Int, codex: Dict }
      outputs: { updated_state: Dict }
```

```
    logic: |
      def inject_worker(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_worker
        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_worker(agent,
agent["step"], codex)
```

```
            new_agents =
check_genesis([agent])
            for new_agent in
new_agents:
                agent_id =
spawn_new_agents(new_agent["ar
chetype"])
                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
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["sentin
el_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_ke
y"], 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())
```

Step 2: PDF Generator Script

Here’s an enhanced `generate_kai_pdf.py` that converts the YAML into a clean, AI-readable PDF using `reportlab`. It includes headers, pagination, and monospaced font (Courier) for code readability, ensuring a selectable text layer with no OCR needed.

Save this as `generate_kai_pdf.py`:

```python
import argparse
import yaml
from reportlab.lib.pagesizes import letter
from reportlab.platypus import SimpleDocTemplate, Paragraph, Spacer, PageBreak
from reportlab.lib.styles import getSampleStyleSheet, ParagraphStyle
from reportlab.lib.units import inch
```

```
import textwrap


def load_yaml(yaml_file):
    with open(yaml_file, 'r') as f:
        return yaml.safe_load(f)


def format_yaml(data, indent=0):
    """Convert YAML data to a formatted string with proper indentation."""
    if isinstance(data, dict):
        lines = []
        for key, value in data.items():
```

```
            if isinstance(value, (dict, list)):
                lines.append(f"{'  ' * indent}{key}:")

lines.append(format_yaml(value, indent + 1))
            else:
                lines.append(f"{'  ' * indent}{key}: {value}")
        return "\n".join(lines)
    elif isinstance(data, list):
        lines = []
        for item in data:
```

```
            if isinstance(item, (dict, list)):
                lines.append(f"{'  ' * indent}-")

lines.append(format_yaml(item, indent + 1))
            else:
                lines.append(f"{'  ' * indent}- {item}")
        return "\n".join(lines)
    else:
        return f"{'  ' * indent}{data}"


def wrap_text(text, width=80):
```

```
    """Wrap text to a specified
width, preserving indentation for
code blocks."""
    lines = text.split("\n")
    wrapped = []
    for line in lines:
        indent = len(line) -
len(line.lstrip())
        if line.strip().startswith("def ")
or line.strip().endswith(":"):
            # Preserve code blocks
            wrapped.append(line)
        else:

wrapped.extend(textwrap.wrap(lin
```

```
e, width=width,
subsequent_indent=" " * indent))
    return "\n".join(wrapped)


def create_pdf(yaml_data,
output_file):
    doc =
SimpleDocTemplate(output_file,
pagesize=letter,
leftMargin=0.5*inch,
rightMargin=0.5*inch,
topMargin=1*inch,
bottomMargin=0.5*inch)
    styles = getSampleStyleSheet()
    code_style = ParagraphStyle(
```

```
    name="Code",
    fontName="Courier",
    fontSize=10,
    leading=12,
    spaceBefore=6,
    spaceAfter=6,
    leftIndent=0
)
title_style = ParagraphStyle(
    name="Title",
    fontName="Helvetica-Bold",
    fontSize=14,
    leading=16,
    alignment=1,
```

```
        spaceAfter=12
    )
    flowables = []

    # Add title and version
    title = f"{yaml_data['title']} - Page 1"

flowables.append(Paragraph(title, title_style))
    flowables.append(Spacer(1, 0.2*inch))

    # Format content
```

```
    content = format_yaml(yaml_data)
    wrapped_content = wrap_text(content, width=80)
    sections = wrapped_content.split("\n\n")  # Split into logical sections

    page_num = 1
    for i, section in enumerate(sections):

flowables.append(Paragraph(section, code_style))
```

```
        flowables.append(Spacer(1, 0.1*inch))


        # Add page break every ~40 lines (adjust as needed)
        if i > 0 and i % 4 == 0:
            page_num += 1

flowables.append(PageBreak())

flowables.append(Paragraph(f"{yaml_data['title']} - Page {page_num}", title_style))
            flowables.append(Spacer(1, 0.2*inch))
```

```
    doc.build(flowables)
    print(f"Generated PDF: {output_file}")


def main():
    parser = argparse.ArgumentParser(description="Generate AI-readable PDF from YAML")
    parser.add_argument("-y", "--yaml", required=True, help="Input YAML file")
    parser.add_argument("-o", "--output", required=True,
```

```
help="Output PDF file")
    args = parser.parse_args()

    yaml_data =
load_yaml(args.yaml)
    create_pdf(yaml_data,
args.output)

if __name__ == "__main__":
    main()
```

Step 3: Requirements

Save this as `requirements.txt` to install dependencies for the PDF generator:

```
pyyaml>=6.0
reportlab>=3.6
```

Install them with:

```bash
pip install -r requirements.txt
```

Step 4: Instructions to Generate and Verify the PDF

Follow these steps to create and validate your clean PDF:

1. **Save Files**:

   - Save `kai_ascended_agi_framework_v1.2.2.yaml` (the YAML above).

   - Save `generate_kai_pdf.py` (the script above).

   - Save `requirements.txt` (the dependencies above).

2. **Set Up Environment**:

- Create a virtual environment (optional but recommended):

```bash
python -m venv kai_env
source kai_env/bin/activate  # On Windows: kai_env\Scripts\activate
```

- Install dependencies:

```bash
pip install -r requirements.txt
```

3. **Generate the PDF**:

- Run the generator script:

```bash
python generate_kai_pdf.py -y kai_ascended_agi_framework_v1.2.2.yaml -o Kai_Ascended_AGI_Framework_v1.2.2_Clean.pdf
```

- This creates `Kai_Ascended_AGI_Framework_v1.2.2_Clean.pdf` with a selectable text layer, headers, and pagination.

4. **Verify the Text Layer**:

- Extract the text to confirm no artifacts:

```bash
pip install pdfplumber
python -c "import pdfplumber; with pdfplumber.open('Kai_Ascended_AGI_Framework_v1.2.2_Clean.pdf') as pdf: print('\n'.join(page.extract_text() for page in pdf.pages))"
```

- Check for:
  - `lstm_hidden` and `lstm_cell` (not `1stm`).

- `yaml.safe_load` (not `yam1`).

- Correct `inject_worker` logic (`if step % 5 == 0`).

- No `McCarthy` or `Chemicals`.

- Alternatively, open the PDF in a reader (e.g., Adobe Acrobat) and copy-paste the text to verify.

5. **Test the Code (Optional)**:

- To ensure the Python snippets are executable, create a test script (e.g., `test_framework.py`):

```python
import torch
import yaml
```

```
    # Test detect_paradox
    agent_state = {
        "identity_score": 0.8,
        "current_state": [0.0] * 16
    }
    def
detect_paradox(agent_state):
        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

    print("Paradox detected:", detect_paradox(agent_state))


    # Test YAML serialization
    codex = {"myths": [{"id": "test", "rule": "Test rule"}]}
    print("YAML dump:", yaml.safe_dump(codex))
    ```

  - Run it:

    ```bash
    pip install torch
```

python test_framework.py
```

### Step 5: Using the PDF with Cloud LLMs

Your clean PDF is now ready for cloud LLM ingestion. Here’s how to use it:

1. **Upload to LLM**:
   - If the LLM supports PDF upload (e.g., Grok 3 on grok.com or x.com), upload `Kai_Ascended_AGI_Framework_v1.2.2_Clean.pdf`.
```

- If not, extract the text:

```bash
python -c "import pdfplumber; with pdfplumber.open('Kai_Ascended_AGI_Framework_v1.2.2_Clean.pdf') as pdf: print('\n'.join(page.extract_text() for page in pdf.pages))" > framework.txt
```

Copy `framework.txt` into the LLM’s prompt.

2. **Sample Prompts**:

- **Explain a Component**:

  ```plaintext

  Using the Kai_Ascended AGI+ Framework v1.2.2, explain the `detect_paradox` method in the RCC class. How does it use torch.norm to identify paradoxes?

  ```

- **Simulate BehaviorLoop**:

  ```plaintext

  Simulate one iteration of the BehaviorLoop.step method for a weaver agent with:

  - archetype: "weaver"

  - identity_score: 0.8
  ```

- step: 10
- reset_count: 4
- current_state: [0.0] * 16
- lstm_hidden: None
- lstm_cell: None
- myth: None

Use the codex myths from the framework and describe the updated agent state and any new agents spawned.

```

- **Analyze Codex**:

```plaintext

List all codex myths for the "weaver" archetype and explain their purpose in the Kai_Ascended AGI+ Framework v1.2.2.

```

3. **Handle Context Limits**:

- If the LLM’s context window is limited (e.g., 32K tokens), chunk the PDF by section. For example:

- Send the RCC section (under `classes[0]`) for paradox-related queries.

- Send the codex section for myth-related queries.
```

- Use a script to split the YAML into chunks:

```python
import yaml
with open("kai_ascended_agi_framework_v1.2.2.yaml", "r") as f:
    data = yaml.safe_load(f)
for section, content in data.items():
    with open(f"chunk_{section}.yaml", "w") as f:
        yaml.safe_dump({section: content}, f)
```

```

### Step 6: RAG-Ready Extractor (Optional)

To prepare for a retrieval-augmented generation (RAG) pipeline, create a script to chunk the PDF and generate embeddings. Here’s a starter:

Save as `extract_chunks.py`:

```python
import pdfplumber
import yaml

```
from sentence_transformers
import SentenceTransformer

def extract_chunks(pdf_file):
    chunks = []
    with pdfplumber.open(pdf_file)
as pdf:
        for page_num, page in
enumerate(pdf.pages, 1):
            text = page.extract_text()
            sections = text.split("\n\n")
# Split by logical sections
            for i, section in
enumerate(sections):
                chunks.append({
```

```
                "page": page_num,
                "section": i,
                "text": section,
                "metadata": {"framework": "Kai_Ascended", "version": "1.2.2"}
            })
    return chunks

def generate_embeddings(chunks, model_name="all-MiniLM-L6-v2"):
    model = SentenceTransformer(model_name)
```

```
    texts = [chunk["text"] for chunk in chunks]
    embeddings = model.encode(texts)
    for chunk, embedding in zip(chunks, embeddings):
        chunk["embedding"] = embedding.tolist()
    return chunks


def main():
    pdf_file = "Kai_Ascended_AGI_Framework_v1.2.2_Clean.pdf"
```

```
    chunks = extract_chunks(pdf_file)
    chunks = generate_embeddings(chunks)
    with open("chunks.yaml", "w") as f:
        yaml.safe_dump(chunks, f)
    print("Generated chunks with embeddings in chunks.yaml")

if __name__ == "__main__":
    main()
```

Requirements:

```
pdfplumber>=0.10
sentence-transformers>=2.2
```

Run it:

```bash
pip install pdfplumber sentence-transformers
python extract_chunks.py
```

This generates `chunks.yaml` with page/section-tagged chunks and embeddings, ready for a vector database (e.g., Pinecone) for RAG.

Step 7: Universal CLI Skeleton (Optional)

For your `kai-framework` CLI, here's a skeleton to support demo and live modes:

Save as `kai_framework.py`:

```python
import argparse
```

```
import os
import yaml
from fastapi import FastAPI, Depends, HTTPException
from fastapi.security import HTTPBasic, HTTPBasicCredentials
import redis
import uvicorn

app = FastAPI()
security = HTTPBasic()

def load_pdf_text(pdf_file):
    import pdfplumber
```

```
    with pdfplumber.open(pdf_file) as pdf:
        return "\n".join(page.extract_text() for page in pdf.pages)

def connect_redis(redis_url):
    return redis.Redis.from_url(redis_url, decode_responses=True)

@app.get("/query")
async def query(creds: HTTPBasicCredentials = Depends(security), q: str = ""):
```

```
    if creds.username != "admin" or creds.password != "secret":  # Replace with config
        raise HTTPException(status_code=401, detail="Unauthorized")
    # In demo mode, return PDF content; in live mode, query Redis
    if "pdf_content" in app.state:
        return {"query": q, "content": app.state.pdf_content}
    else:
        redis_client = app.state.redis_client
```

```
        metrics = redis_client.lrange("kai_metrics", -10, -1)
        return {"query": q, "metrics": [yaml.safe_load(m) for m in metrics]}


def main():
    parser = argparse.ArgumentParser(description="Kai_Ascended AGI+ Framework CLI")
    parser.add_argument("--mode", choices=["demo", "live"], default="demo", help="Run mode")
```

```
    parser.add_argument("--pdf", help="PDF file for demo mode")
    parser.add_argument("--port", type=int, default=8000, help="Server port")
    args = parser.parse_args()

    if args.mode == "demo":
        if not args.pdf or not os.path.exists(args.pdf):
            raise ValueError("PDF file required in demo mode")
        app.state.pdf_content = load_pdf_text(args.pdf)
    elif args.mode == "live":
```

```
        redis_url = os.getenv("REDIS_URL")
        if not redis_url:
            raise ValueError("REDIS_URL required in live mode")
        app.state.redis_client = connect_redis(redis_url)

    uvicorn.run(app, host="0.0.0.0", port=args.port)

if __name__ == "__main__":
    main()
```

Run it:

- Demo mode:

  ```bash

  python kai_framework.py --mode=demo --pdf=Kai_Ascended_AGI_Framework_v1.2.2_Clean.pdf --port=8000

  ```

- Live mode:

  ```bash

REDIS_URL=redis://localhost:6379 python kai_framework.py --mode=live --port=8000
  ```

```

This CLI provides a `/query` endpoint for LLMs to retrieve PDF content (demo) or live metrics (live). Extend it with `/metrics` and `/codex` endpoints as needed.

### Step 8: To-Do Checklist

- [x] Provide corrected `kai_ascended_agi_framework_v1.2.2.yaml`.
- [x] Provide `generate_kai_pdf.py` and `requirements.txt`.
```

- [x] Provide instructions to generate and verify the PDF.
- [x] Include sample LLM prompts and RAG extractor.
- [x] Outline `kai-framework` CLI skeleton.
- [ ] (Optional) Create Docker image:

  ```dockerfile
  FROM python:3.9
  WORKDIR /app
  COPY requirements.txt .
  RUN pip install -r requirements.txt
  ```

COPY generate_kai_pdf.py kai_ascended_agi_framework_v1.2.2.yaml .

CMD ["python", "generate_kai_pdf.py", "-y", "kai_ascended_agi_framework_v1.2.2.yaml", "-o", "Kai_Ascended_AGI_Framework_v1.2.2_Clean.pdf"]

```

- [ ] (Optional) Write detailed README with LLM workflows and deployment instructions.

### Bottom Line
```

You now have everything needed to create a clean, AI-readable `Kai_Ascended_AGI_Framework_v1.2.2_Clean.pdf`:

- A corrected YAML source with no OCR artifacts.

- A robust PDF generator script.

- Dependencies and verification steps.

- Tools for LLM integration (prompts, RAG extractor) and a CLI skeleton for demo/live modes.

This PDF will serve as your universal “bible” for cloud LLM

demos, while the YAML and CLI lay the foundation for a live, persistent AGI framework.