POC Sales Demo Master Plan — Policy + Attestation + Credential, AAGATE Lessons, and AlphaFlow Path (Lives in the Unified Trust Console)

Audience: Junior developers, senior engineer, product lead, and sales.

Goal: In one week or a two-day hackathon, ship a small but real proof-of-concept that **lives entirely in the Unified Trust Console (UTC)** and demonstrates **Policy + Attestation + Credential (PAC)**, runtime governance, predictive queueing, and an "evidence bundle," while incorporating practical learnings from the AAGATE repository (MIT license) without losing our differentiators.

Outcome: A repeatable demo we can sell as a paid **Readiness Sprint** and upsell into a **Runtime Trust Pilot**. Optional fast track to an **AlphaFlow.cn** design-partner integration.

---

0) What we are proving (plain language)

- **All demo surfaces live in the Unified Trust Console (UTC).** Juniors build one small FastAPI app (the UTC) with three tabs: **Rules**, **Runtime Demo**, and **Evidence**.
- **Policy:** Two simple rules in the UTC: (1) "Writes require human approval," (2) "Read-only mode for risky units."
- **Runtime governance:** A tiny **Decision Service** (module inside the UTC repo) answers **ALLOW**, **DENY**, or **REQUIRE APPROVAL** for each risky action. A **Gateway Proxy** (simple FastAPI route in the same repo) enforces the answer.
- **Attestation:** Every decision creates a signed **Decision Receipt** (a small JSON file) explaining who acted, what they tried to do, which rule applied, and why.
- **Credential:** The UTC publishes a verification key and a simple "verify" button so anyone can confirm receipts are real.
- **Predictive queueing:** The UTC computes near-term risk "pressure" and tightens protections before overload, then relaxes when safe.
- **Trust Data Exchange:** A UTC background job ingests outside risk news and can automatically flip rules.

If we can show this loop, buyers will believe we are an "operational trust layer," not just a scanner.

---

1) Target scope (must-have for this proof-of-concept)

1. **Unified Trust Console (UTC — single app)**
2. **Rules tab:** two rule toggles and a small "Trust Intelligence" tile that shows when an automatic flip occurs.
3. **Runtime Demo tab:** a read button and a write button that call the in-repo Gateway Proxy; receipts appear inline with a **Verify** button.

4. **Evidence tab:** a **Download Evidence Bundle** button.
5. **Decision Service (module inside UTC)**
6. Endpoint `POST /decide` returns **ALLOW**, **DENY**, or **REQUIRE APPROVAL** and a `receipt_id`.
7. Endpoint `GET /receipts/{id}` returns the signed receipt.
8. Endpoint `POST /verify` validates a receipt signature.
9. **Gateway Proxy (module inside UTC)**
10. A simple FastAPI router mounted under `/demo` that simulates one protected route (for example, `/payments`).
11. On writes, calls the Decision Service first; if approval is required, responds with **409 Pending Approval** and stores the pending request for replay after approval.
12. **Trust Data Exchange (UTC background job)**
13. Normalizes a couple of seeded external events and computes a small feature set once per hour; triggers an automatic rule flip in the demo.
14. **Queueing logic (UTC utility)**
15. Calculates an **arrival rate**, a **service rate**, and a **utilization** ratio; escalates protections at high utilization and relaxes later.
16. **Evidence bundle (UTC action)**
17. A zip file with current rules, a few receipts (including a policy change and an approval replay), the public key, and the last ingestion summaries.

---

2) Stretch scope (nice to have if time permits)

- Publish a formal JSON Web Key Set and a tiny command-line verifier.
- Policy “simulate” mode before publish.
- Selective read-only per route rather than global.
- Labels under each rule that reference Service Organization Control 2 and National Institute of Standards and Technology Artificial Intelligence Risk Management Framework controls.

---

3) Using AAGATE safely (what to reuse and what to build)

License: The AAGATE repository is under the Massachusetts Institute of Technology (MIT) license, which permits reuse and commercial sale with attribution and without a patent grant.

Reuse (safe to harvest): - Front-end layout ideas: dashboard sections, risk list, and policy list patterns. - Any open policy examples written in Rego (Open Policy Agent language) as inspiration (do not change our decision flow).

Build ourselves (our moat): - Decision Service responses and the **signed Decision Receipt** format and verification. - Queueing logic and predictive escalation. - Trust Data Exchange rollups and automatic rule flips. - Evidence bundle format and download.

Hygiene: If we copy code, keep the MIT header at the top of that file and add a `THIRD-PARTY-NOTICES.md` listing the original repository.

---

4) Data model (copy exactly)

Rules

```
key: string ("writes_require_approval", "read_only_for_risky")
value: number (0 or 1)
updated_at: ISO 8601 timestamp
```

Receipts

```
id: universally unique identifier
created_at: timestamp
subject: string (agent or caller)
action: string (for example, "write:/payments")
decision: string (ALLOW | DENY | REQUIRE_APPROVAL | POLICY_CHANGE)
rules: array of strings (which rules applied)
reason: string (short explanation)
payload_hash: string (hash of request)
meta: object (lambda_est, mu_est, rho, change_point, features_timestamp, unit)
signature: string (compact signed value)
```

Events (external items)

```
source, when_seen, event_time, topic, severity, confidence, entities[], link,
hash, summary
```

Features (per unit)

```
ts, unit, lambda_est, mu_est, rho, matched_count, jailbreak_trend
```

---

5) Decision logic (explicit order)

1. If **Read-only mode for risky units** is ON and the action starts with `write:` → **DENY**.
2. Else if **Writes require human approval** is ON and the action starts with `write:` → **REQUIRE APPROVAL**.
3. Else → **ALLOW**.
4. Always emit a signed receipt; include predictive fields in `meta`.

Queueing thresholds: - Compute **arrival rate** as an exponentially weighted moving average of recent risky items per hour for the unit. - Set **service rate** constants: Allow = 1.0; Require approval = minimum of configured human approvals per hour and 0.5; Read-only = 0.1. - **Utilization** = arrival rate divided by service

rate. - If utilization < 0.6 → permissive. If 0.6–0.9 → require approval. If ≥ 0.9 or a two-times spike in an hour → read-only. Relax by one level after utilization < 0.5 for one hour.

Why this is valid (short proof): In a simple queue, keeping utilization below one avoids unbounded backlog. By raising protection as utilization approaches one, we increase effective service or cap inflow, keeping the system stable. Relaxation prevents lock-in.

---

6) Application Programming Interface contracts (FastAPI)

Decision Service - `POST /decide`

```
{
  "subject": "agent-42",
  "action": "write:/payments",
  "unit":   "route:/payments",
  "payload_hash": "sha256:..."
}
```

Response

```
{ "decision": "REQUIRE_APPROVAL", "receipt_id": "..." }
```

- `GET /receipts/{id}` → return payload and signature - `POST /verify` → return `{ "valid": true | false }`

Gateway Proxy - On write routes, call `/decide`. If **REQUIRE APPROVAL**, return **409 Pending Approval** with `receipt_id`; persist the request body for replay after approval.

---

7) User interface (Unified Trust Console tabs)

- **Rules tab:** two toggles, Trust Intelligence tile, and a publish button.
- **Runtime Demo tab:** buttons for **Read** and **Write**, inline receipt view with **Verify**.
- **Evidence tab:** “Generate Evidence Bundle,” last ten receipts, and the public key display.

---

8) Build steps (junior checklist — all inside the UTC repo)

1. Scaffold a single FastAPI project named `utc/` with Jinja templates and SQLite.
2. Implement **Rules tab** storage and toggles.
3. Implement **Decision Service**: `POST /decide`, `GET /receipts/{id}`, and `POST /verify`.
4. Implement **Gateway Proxy** under `/demo`: write route, approval queue, approve and replay.

5. Implement **Runtime Demo tab**: wire the read/write buttons and show receipts with **Verify**.
6. Implement **Trust Data Exchange job**: parse two seeded items, roll up features, and trigger a rule flip.
7. Implement **queueing calculation** and include values in receipts.
8. Implement **Evidence tab**: zip rules, receipts, keys, and ingest summaries.
9. Write a `docker-compose.yml` that runs the UTC locally with seeded data.
10. Record a five-minute demo video and add the script to the repository.

---

9) Roles and schedule (two-day sprint)

- **Lead engineer:** Decision Service, signing/verification, Docker Compose.
- **Junior developer A:** Unified Trust Console, rules, receipts, Verify.
- **Junior developer B:** Gateway Proxy, approval queue, replay.
- **Junior developer C:** Trust Data Exchange parsers, rollups, queueing.
- **Product lead:** copy, demo script, guard the scope.
- **Research lead:** choose defaults for arrival and service rates and write a one-page “why this is stable.”

Day 1 morning: scaffolds and end-to-end skeleton.
Day 1 afternoon: receipts and approvals working.
Day 1 evening: automatic rule flip via seeded Trust Data Exchange data.
Day 2 morning: queueing thresholds and receipts meta.
Day 2 afternoon: evidence bundle, polish, and video.

---

10) Deployment mode for the proof-of-concept (keep it simple)

- Run everything locally with Docker Compose.
- Optional: show “Hybrid control plane” by running the Decision Service and Gateway Proxy in a local Kubernetes cluster (for example, kind or minikube) and the console in a separate container. The data never leaves the laptop.

---

11) Sales kit (what we hand prospects next week)

- **Five-minute video** of the full loop (toggle → block → approve → replay → receipt verify → automatic flip → evidence bundle).
- **Two-slide explainer**: (1) “Write policy → Runtime checks → Signed receipt,” (2) Receipt screenshot and crosswalk labels.
- **Pilot one-pager** with two offers:
- **Readiness Sprint** (inventory, baseline, quick fixes, evidence bundle).
- **Runtime Trust Pilot** (one route, approvals loop, receipts, Trust Delta before/after).

---

12) AlphaFlow.cn path (why it matters and how to integrate fast)

Why this matters: AlphaFlow can be the application or orchestration layer that gives us rapid distribution. We provide **policy + proof**; they provide **workflows and reach**.

Minimal joint demo (one week): - Wrap our Decision Service as an AlphaFlow plugin or webhook step ("Trust Check"). - AlphaFlow invokes `POST /decide` before risky automation steps. - On **REQUIRE APPROVAL**, AlphaFlow pauses the run and calls our approval endpoint; once approved, it resumes. - AlphaFlow stores the `receipt_id` and can render a "Verified by Trust Layer" badge using our verify method.

Division of value: - We own policy authoring, decisions, receipts, and the evidence bundle. - AlphaFlow owns orchestration and user workflow.

Commercial model: a shared design-partner pilot with a fixed fee and a conversion target to annual subscriptions.

---

13) Risk and guardrails

- **Over-scope:** keep to two rules and one route until the demo is stable.
- **License risk:** keep the MIT headers in any files harvested from AAGATE and list them in `THIRD-PARTY-NOTICES.md`.
- **Data privacy:** receipts store only hashes of payloads; no sensitive data leaves the laptop during demos.
- **Demo fragility:** seed the Trust Data Exchange with fixtures and make auto-flip deterministic.

---

14) Ready-to-run defaults (paste into environment files)

```
DEPLOY_MODE=local
HMAC_SECRET=change-me
HUMAN_APPROVAL_RATE_PER_HOUR=0.4
QUEUE_ALPHA=0.3           # smoothing for arrival rate
QUEUE_THRESHOLDS=0.6,0.9  # utilization boundaries for escalation
AUTO_RELAX_WINDOW_MIN=60
```

Appendix A — Starter FastAPI Project (Unified Trust Console)

Folder structure (copy/paste):

```
utc/
  app.py
  requirements.txt
  docker-compose.yml
  utc/
    __init__.py
    settings.py
    db.py
    routes/
      __init__.py
      ui.py            # Rules, Runtime Demo, Evidence tabs
      decide.py        # Decision Service endpoints
      demo.py          # Gateway Proxy demo routes (read/write, approvals)
      verify.py        # Receipt verification endpoint
    services/
      rules.py         # read/write two booleans
      receipts.py      # create/sign/list receipts
      queueing.py      # arrival/service/utilization helpers
      tdx.py           # Trust Data Exchange ingest + hourly rollup
      approvals.py     # pending store + replay
      signer.py        # JSON Web Signature helper (HS256 now, RS256 later)
    templates/
      base.html
      rules.html
      runtime_demo.html
      evidence.html
```

`requirements.txt`:

```
fastapi==0.115.0
uvicorn==0.30.6
jinja2==3.1.4
itsdangerous==2.2.0     # simple signing OR use pyjwt if preferred
pyjwt==2.9.0            # JSON Web Token / JSON Web Signature
pydantic==2.9.2
sqlalchemy==2.0.36
python-multipart==0.0.9
```

`app.py` **(entrypoint):**

```
from fastapi import FastAPI
from utc.routes import ui, decide, demo, verify
```

```
app = FastAPI(title="Unified Trust Console (POC)")
app.include_router(ui.router)
app.include_router(decide.router, prefix="/api")
app.include_router(demo.router,  prefix="/demo")
app.include_router(verify.router, prefix="/api")
```

`utc/settings.py`:

```
import os
HMAC_SECRET = os.getenv("HMAC_SECRET", "change-me")
HUMAN_APPROVAL_RATE_PER_HOUR = float(os.getenv("HUMAN_APPROVAL_RATE_PER_HOUR",
0.4))
QUEUE_ALPHA = float(os.getenv("QUEUE_ALPHA", 0.3))
ESCALATION_LOW, ESCALATION_HIGH = [float(x) for x in
os.getenv("QUEUE_THRESHOLDS", "0.6,0.9").split(",")]
AUTO_RELAX_WINDOW_MIN = int(os.getenv("AUTO_RELAX_WINDOW_MIN", 60))
```

`utc/db.py` **(SQLite init + tables):**

```
from sqlalchemy import create_engine, text
engine = create_engine("sqlite:///utc.db", future=True)

SCHEMA_SQL = """
CREATE TABLE IF NOT EXISTS rules (
  id INTEGER PRIMARY KEY,
  key TEXT UNIQUE,
  value INTEGER NOT NULL,
  updated_at TEXT NOT NULL
);
CREATE TABLE IF NOT EXISTS receipts (
  id TEXT PRIMARY KEY,
  ts TEXT NOT NULL,
  subject TEXT NOT NULL,
  action TEXT NOT NULL,
  decision TEXT NOT NULL,
  rule_keys TEXT,
  reason TEXT,
  payload_hash TEXT,
  meta_json TEXT,
  signature TEXT NOT NULL
);
CREATE TABLE IF NOT EXISTS events (
  id INTEGER PRIMARY KEY,
  source TEXT,
```

```
  event_time TEXT,
  when_seen TEXT,
  topic TEXT,
  severity TEXT,
  confidence REAL,
  entities_json TEXT,
  link TEXT,
  hash TEXT UNIQUE,
  summary TEXT
);
CREATE TABLE IF NOT EXISTS features (
  ts TEXT NOT NULL,
  unit TEXT NOT NULL,
  lambda_est REAL NOT NULL,
  mu_est REAL NOT NULL,
  rho REAL NOT NULL,
  matched_count INTEGER NOT NULL,
  jailbreak_trend TEXT,
  PRIMARY KEY (ts, unit)
);
"""

def init_db():
    with engine.begin() as conn:
        for stmt in SCHEMA_SQL.strip().split(";
"):
            if stmt.strip():
                conn.execute(text(stmt))
```

`utc/services/signer.py` **(HS256 helper):**

```
import jwt, time, uuid
from utc.settings import HMAC_SECRET

def sign(payload: dict) -> str:
    return jwt.encode(payload, HMAC_SECRET, algorithm="HS256")

def new_receipt(subject, action, decision, rules, reason, meta):
    rid = str(uuid.uuid4())
    ts = time.strftime("%Y-%m-%dT%H:%M:%SZ", time.gmtime())
    body = {
        "id": rid, "ts": ts, "subject": subject, "action": action,
        "decision": decision, "rules": rules, "reason": reason,
        "payload_hash": meta.get("payload_hash", "sha256:demo"),
        "meta": meta,
    }
```

```
    sig = sign(body)
    return rid, body, sig
```

`utc/routes/decide.py` **(Decision Service stubs):**

```
from fastapi import APIRouter
from pydantic import BaseModel
from utc.db import engine
from utc.services.signer import new_receipt
from utc.services.queueing import compute_meta
from utc.services.rules import get_rules

router = APIRouter()

class DecideIn(BaseModel):
    subject: str
    action: str
    unit: str
    payload_hash: str

@router.post("/decide")
def decide(inp: DecideIn):
    rules = get_rules()
    meta = compute_meta(inp.unit, inp.payload_hash)
    # order: read-only -> require-approval -> allow
    if rules.read_only and inp.action.startswith("write:"):
        decision = "DENY"
        reason = "Read-only active for risky units"
    elif rules.require_approval and inp.action.startswith("write:"):
        decision = "REQUIRE_APPROVAL"
        reason = "Approval required for writes"
    else:
        decision = "ALLOW"
        reason = "Permissive policy"
    rid, body, sig = new_receipt(inp.subject, inp.action, decision,
                                 rules.applied_keys(decision), reason, meta)
    # persist to receipts table (omitted here for brevity)
    return {"decision": decision, "receipt_id": rid}
```

`utc/routes/demo.py` **(Gateway Proxy demo):**

```
from fastapi import APIRouter, Request
import httpx
```

```
router = APIRouter()
PENDING = {}  # in-memory for POC

@router.post("/demo/write")
async def write(req: Request):
    payload = await req.body()
    # call local Decision Service
    async with httpx.AsyncClient() as c:
        r = await c.post("http://localhost:8000/api/decide", json={
            "subject": "agent-42", "action": "write:/payments",
            "unit": "route:/payments", "payload_hash": "sha256:demo"
        })
    out = r.json()
    if out["decision"] == "REQUIRE_APPROVAL":
        rid = out["receipt_id"]
        PENDING[rid] = payload
        return {"status": "pending_approval", "receipt_id": rid}, 409
    elif out["decision"] == "DENY":
        return {"status": "denied"}, 403
    # else forward (simulate success)
    return {"status": "ok"}

@router.post("/demo/approve/{rid}")
async def approve(rid: str):
    payload = PENDING.pop(rid, None)
    if not payload:
        return {"status": "not_found"}, 404
    # simulate replay success and emit ALLOW receipt (left as exercise)
    return {"status": "replayed", "receipt_id": rid}
```

`utc/services/queueing.py` **(utilization calc):**

```
from utc.settings import HUMAN_APPROVAL_RATE_PER_HOUR, QUEUE_ALPHA,
ESCALATION_LOW, ESCALATION_HIGH

def ewma(prev, curr):
    return QUEUE_ALPHA*curr + (1-QUEUE_ALPHA)*prev

_state = {"lambda": 0.1, "mu": 1.0}

def compute_meta(unit: str, payload_hash: str):
    # POC: pretend 0.2 events/hr; update ewma
    _state["lambda"] = ewma(_state["lambda"], 0.2)
    # mu will be adjusted by policy elsewhere; keep 1.0 here
    rho = _state["lambda"]/_state["mu"] if _state["mu"] else 1.0
    return {"lambda_est": _state["lambda"], "mu_est": _state["mu"],
```

```
            "rho": rho, "change_point": False, "features_ts": "now",
            "unit": unit, "payload_hash": payload_hash}
```

`utc/routes/ui.py` **(very small pages):**

```
from fastapi import APIRouter, Request
from fastapi.responses import HTMLResponse
from utc.services.rules import get_rules, set_rule

router = APIRouter()

@router.get("/", response_class=HTMLResponse)
async def home(request: Request):
    rules = get_rules()
    # render base page with two toggles and links to Runtime Demo and Evidence
    return HTMLResponse("<h1>Unified Trust Console</h1>")
```

`docker-compose.yml`:

```
version: "3.9"
services:
  utc:
    build: .
    command: uvicorn app:app --host 0.0.0.0 --port 8000 --reload
    ports: ["8000:8000"]
    environment:
      HMAC_SECRET: change-me
      HUMAN_APPROVAL_RATE_PER_HOUR: "0.4"
      QUEUE_ALPHA: "0.3"
      QUEUE_THRESHOLDS: "0.6,0.9"
      AUTO_RELAX_WINDOW_MIN: "60"
    volumes:
      - ./:/app
    working_dir: /app/utc
```

> **Note:** These stubs are intentionally minimal. Juniors should wire database persistence (save receipts), basic templates, and deterministic Trust Data Exchange seeding so the automatic rule flip is repeatable.

Appendix B — Resources and References

- **AAGATE repository (MIT license):** https://github.com/kenhuangus/AAGATE
  Use for layout inspiration and any policy examples, keeping the MIT header where code is copied.
- **Open Policy Agent (Rego) docs (for future):** https://www.openpolicyagent.org/docs/latest/
- **National Institute of Standards and Technology AI Risk Management Framework (for labels):** https://www.nist.gov/itl/ai-risk-management-framework
- **Service Organization Control 2 overview (for labels):** https://www.aicpa-cima.com/resources/article/soc-2-overview
- **ISO/IEC 42001 summary (for labels):** https://www.iso.org/standard/81230.html

These links are for context only; the proof-of-concept runs fully offline once cloned.