# Final Enterprise Project – Deployment & Reliability Runbook (README)

**Goal:** Anyone (teammates, TA, or judge) can set up, run, and recover the multi-agent system in minutes.

**Team 2 Civic Problem Statement**

Financial literacy is not just about numbers; it's about equity, opportunity, and empowerment. In the U.S., entire communities are left behind because they don't have easy access to financial education or trustworthy guidance. Low-income families, students, immigrants, and those living paycheck to paycheck often face the harshest realities: predatory lending, misinformation, and limited awareness of benefits or programs meant to support them. This creates a cycle of inequity, where individuals with the least resources face the most significant challenges.

At the same time, technology has become one of the most universal access points. While financial systems can be intimidating, most people have access to a phone, computer, or basic internet connection. That's why our project uses technology as the bridge, delivering financial literacy and equity through a medium that can reach everyone. By leveraging AI agents, we aim to take concepts once locked behind paywalls, institutions, or jargon-heavy textbooks, and put them into clear, actionable guidance that works for any background.

Our solution is a multi-agent financial literacy assistant designed around equity and accessibility. Instead of a single "one-size-fits-all" voice, our system is built on a main orchestrator agent supported by sub-agent personas, each one a niche financial helper.

The Main Agent (Orchestrator) acts as the conductor, listening to the user's needs, routing questions, and synthesizing responses.

The Sub-Agents each specialize in a critical area where underserved communities often need help most: stock picks and general market research

---

## Check-in 2 Deliverables Checklist (Team – Finance Manager)

**Due:** Thu **Sep 25, 8:00pm ET** (submission email before meeting window)

1. **Final Architecture Diagram** — Update `FinanceAgentArchitecture.drawio` and export to `docs/diagram.png` showing agents, MCP/A2A paths, tools, and ports.

2. **Deployment Tested & Documented** — `docker compose up --build -d` runs **api**, **mcp**, **n8n**, **redis**; `/ui` reachable; `/metrics` shows counters; restart & scale steps included.

3. **Slide Deck (Draft)** — 10–12 slides: problem & users, architecture, agent roles, integrations, demo flow, rate/cost controls, metrics snapshot, risks, next steps.

4. **Budget Plan & Receipts** — Confirm LLM split (e.g., $10 Gemini / $10 OpenAI or chosen plan) and **upload receipts to Bill.com**; include 1-line budget statement in README.

5. **End-to-End Demo Working** — n8n webhook → **orchestrator** → **budget/stock agent via MCP** → tool call (Slack or finance API) → response; structured logs + memory on.

6. **Logging, Errors, Memory** — JSON logs for I/O & errors; retries/backoff; memory store; `/metrics` returns **requests, tokens, cost_usd, p50/p95**.

7. **One-Cycle Cost Proof** — Run `./scripts/run_cycle.sh`; capture the printed **cycle cost** and `/metrics` screenshot; paste snapshot into slide deck.

8. **Submission Email Package** — Include: repo link, Colab link, `docs/diagram.png`, README, slide deck (draft), `/metrics` snapshot after one cycle, budget plan line, **Bill.com receipts confirmation**.

9. **Dry-Run Checklist (15 min)** — Clone → `.env` → compose up → `scripts/smoke.sh` → `run_cycle.sh` (cost delta shows) → throttle test (`REQUESTS_PER_MINUTE`) → budget cap test (set `TOKEN_BUDGET_USD=0.01` to see 429) → quick restart/scale.

10. **Backup Plan (If Anything Breaks)** —

- **LLM keys fail:** enable dry-run/stub mode; use cached responses; demo `/metrics` from prior run.
- **MCP down:** call tools directly from API as fallback; keep interfaces identical.
- **Service crash:** `docker compose logs -f <svc>` → `docker compose restart <svc>`; keep a second API replica (`--scale api=2`).

---

# 0) TL;DR Quickstart

```
# 1) Clone and enter
git clone <REPO_URL> kura-final-project && cd kura-final-project

# 2) Copy env template and fill secrets
cp .env.example .env
# Open .env and set API keys, webhooks, Slack tokens, etc. DO NOT COMMIT .env

# 3) Build & run all services
docker compose up --build -d

# 4) Open the system
http://localhost:8000
# API (LangChain/FastAPI): http://localhost:8000
# MCP server(s): internal; exposed on 8001-8003 (example)
# Admin logs:  docker compose logs -f <service>

# 5) Smoke test (ping)
curl http://localhost:8000/healthz
```

---

# 1) System Overview

**Civic Challenge:**

**Architecture Type:** <hierarchical | shared env | MCP | A2A>
**External Integrations:** <Slack | Google Maps | Financial API | Other>
**Agents:** At least two, each with reasoning (ReAct or Plan&Execute), memory, and tool use.

### Diagram

Attach `docs/diagram.png` (draw.io/Lucid). Must show:

- Agents and their roles
- Tools/APIs/MCP servers
- Communication channels (direct, MCP, or A2A)

---

# 2) Repo Structure

```
.
├─ docker-compose.yml
├─ .env.example            # Copy to .env – keep secrets out of Git
├─ services/
│  ├─ api/                 # LangChain + FastAPI wrapper
│  │  ├─ app.py
│  │  ├─ chains/
│  │  ├─ tools/
│  │  └─ requirements.txt
│  ├─ mcp/                 # Model Context Protocol server(s)
│  │  ├─ server.py         # Registers tools/endpoints exposed via MCP
│  │  ├─ tools/
│  │  └─ requirements.txt
│  └─ n8n/                 # n8n workflow (orchestrations & webhooks)
│     ├─ workflows/
│     │  └─ main.json
│     └─ Dockerfile        # optional; we can use upstream image
├─ data/                   # local dev data (non-PII)
├─ docs/
│  ├─ diagram.png
│  └─ evaluation.md
├─ scripts/
│  ├─ smoke.sh             # health checks and API pings
│  └─ budget_guard.py      # token usage tracker (see §7)
└─ README.md (this file)
```

---

# 3) Prerequisites

- Docker Desktop 4.x or Docker Engine 24+
- docker compose v2
- Git, curl
- Ports available: 5678 (n8n), 8000 (API), 7001 (MCP), 6379 (Redis optional)

---

# 4) Environment Variables & Secrets

> **All secrets live in `.env`. Never commit `.env`.**

Create `.env` from the template:

```
# ——— Core ———
APP_ENV=dev
LOG_LEVEL=INFO

# ——— LLMs (choose at least one) ———
OPENAI_API_KEY=
OPENAI_MODEL=gpt-4o-mini
GEMINI_API_KEY=
GEMINI_MODEL=gemini-1.5-flash

# ——— Rate Limit & Budget ———
# Hard cap: system exits or returns 429 when over budget
TOKEN_BUDGET_USD=20.00
BUDGET_HARD_STOP=true
REQUESTS_PER_MINUTE=60
MAX_TOKENS_PER_CALL=1500

# ——— Slack (if used) ———
SLACK_BOT_TOKEN=
SLACK_SIGNING_SECRET=
SLACK_APP_LEVEL_TOKEN=
SLACK_ALLOWED_CHANNEL=

# ——— External APIs (examples) ———
GOOGLE_MAPS_API_KEY=
FINANCE_API_KEY=
```

```
# ——— MCP Server ———
MCP_HOST=0.0.0.0
MCP_PORT=7001
MCP_ALLOWED_TOOLS=slack,search,finance

# ——— Persistence / Caches ———
REDIS_URL=redis://redis:6379/0
VECTOR_DB_PATH=/data/vector
```

Provide teammates a `.env.example` with the keys above (empty values). Judges should be able to run with demo keys or mocked adapters.

---

# 5) Docker Compose

`docker-compose.yml` (minimal working example):

```
version: "3.9"
services:
  api:
    build: ./services/api
    container_name: api
    env_file: .env
    ports:
      - "8000:8000"
    depends_on:
      - redis
    command: ["python", "app.py"]
    restart: unless-stopped

  mcp:
    build: ./services/mcp
    container_name: mcp
    env_file: .env
    ports:
      - "7001:7001"
    command: ["python", "server.py"]
    restart: unless-stopped

  n8n:
    image: n8nio/n8n:latest
    container_name: n8n
```

```
    env_file: .env
    ports:
      - "5678:5678"
    volumes:
      - ./services/n8n/workflows:/home/node/.n8n
    restart: unless-stopped

  redis:
    image: redis:7-alpine
    container_name: redis
    ports:
      - "6379:6379"
    restart: unless-stopped
```

### Build & Run

```
docker compose up --build -d
```

### Stop & Clean

```
docker compose down     # stop
docker compose down -v  # stop + remove volumes (dev only)
```

### Scale

```
# Scale stateless API replicas for load or demo resilience
docker compose up -d --scale api=2
```

---

# 6) Service Details

## 6.1 API (LangChain + FastAPI)

**Path:** `services/api/app.py`
**Purpose:** Exposes `/invoke`, `/healthz`, and `/metrics`. Wraps LangChain graphs/chains and enforces rate & budget guards.

**Example minimal `app.py`:**

```
from fastapi import FastAPI, HTTPException
import os, time
```

```
from pydantic import BaseModel

app = FastAPI()

BUDGET_USD = float(os.getenv("TOKEN_BUDGET_USD", 20))
MAX_TOKENS = int(os.getenv("MAX_TOKENS_PER_CALL", 1500))

class InvokeReq(BaseModel):
    agent: str
    input: str

@app.get("/healthz")
def healthz():
    return {"status": "ok", "time": time.time()}

@app.post("/invoke")
def invoke(req: InvokeReq):
    # TODO: route to agent via MCP/A2A, enforce token caps, log I/O
    if len(req.input) > 8000:
        raise HTTPException(413, "input too large")
    return {"agent": req.agent, "output": "<stubbed response>", "tokens": 123}
```

## 6.2 MCP Server(s)

**Path:** `services/mcp/server.py`
**Purpose:** Registers tools (Slack, search, finance) and exposes them to agents via MCP. Enforces allow-list via `MCP_ALLOWED_TOOLS`.

**Server responsibilities:**

- Expose tool schemas (inputs/outputs) for each integration
- Authentication: read tokens from env only
- Structured logs: every call has `trace_id`, latency, status

**Example skeleton:**

```
# server.py (skeleton)
import os
from fastapi import FastAPI
```

```
app = FastAPI()
ALLOWED = set(os.getenv("MCP_ALLOWED_TOOLS", "").split(","))

@app.get("/tools")
def list_tools():
    return {"tools": sorted([t for t in ALLOWED if t])}

@app.post("/tool/slack.post_message")
def slack_post_message(payload: dict):
    # read SLACK_BOT_TOKEN from env
    # validate channel in SLACK_ALLOWED_CHANNEL
    # call Slack API, return message ts
    return {"ok": True, "ts": "123.456"}
```

### 6.3 n8n (Orchestration)

**Path:** `services/n8n/workflows/main.json`
**Purpose:** Receives external triggers (webhook/Slack), fans out to API or MCP tools, handles retries.

**Runbook:**

- Import `main.json` into n8n UI → set credentials via env
- Webhook URL → feed into `/invoke` with `{agent, input}`
- On failure (>= 500), backoff and retry up to 3x

---

## 7) Rate Limits & Cost Management

**Constraints:** Team budget ≤ **$20** for LLM tokens.

**Controls:**

1. **Hard budget guard** (`scripts/budget_guard.py`): sums estimated token cost; when `BUDGET_HARD_STOP=true` and projected spend > `TOKEN_BUDGET_USD`, return `429 BudgetExceeded`.

2. **Max tokens per call** (`MAX_TOKENS_PER_CALL`): keep under 1500 (demo-safe).
3. **RPM throttle** (`REQUESTS_PER_MINUTE`): token bucket limiter at API ingress.
4. **Prompt caching**: memoize final tool plans & responses for repeated demo prompts.
5. **Short prompts**: system prompt compressed; tool outputs summarized.
6. **Batching**: where possible, group tool queries.
7. **Fallback models**: prefer `gpt-4o-mini`/`gemini-flash` for planning; escalate to larger only when required.

**Monitoring:**

- `/metrics` endpoint emits: `tokens_in`, `tokens_out`, `cost_usd`, `latency_ms`, `error_rate`.
- `docker compose logs -f api | mcp | n8n` during demo.

---

# 8) Logging, Error Handling, & Evaluation

**Structured Logs (JSON):** every request logs `ts, trace_id, agent, tool, latency_ms, tokens, cost_usd, status`.

**Retries & Fallbacks:**

- `429/5xx`: exponential backoff (200ms, 400ms, 800ms; max 3 attempts)
- Tool failure → fallback to alternate tool or summarization-only mode

**Evaluation Metrics:**

- **Latency:** p50/p95 service times per agent
- **Accuracy/Task Success:** % tasks completed on test prompts

- **Reliability:** error rate < 2% on 20-prompt suite

See `docs/evaluation.md` for results.

---

# 9) Running the Demo (Run-of-Show)

1. **Pre-flight (5 min):**
    - `docker compose up -d` and open logs
    - Check `/healthz` for API & MCP
    - Trigger n8n test webhook → expect 200
2. **Live Walkthrough:**
    - Show diagram, then run a real task end-to-end
    - Display structured logs (one success, one handled retry)
3. **Failure Drill:**
    - Kill one `api` replica → show auto-recovery (`--scale api=2`)
    - Exceed RPM → demonstrate throttling message
4. **Q&A:**
    - Rate-limit strategy, budget guard, and restart steps

---

# 10) Troubleshooting

| Symptom | Likely Cause | Fix |
| --- | --- | --- |
| `401 Unauthorized` to Slack | Missing/invalid `SLACK_BOT_TOKEN` | Set token in `.env`, restart `n8n`/`mcp` |

| | | |
|---|---|---|
| `429 BudgetExceeded` | Token budget cap hit | Lower `MAX_TOKENS_PER_CALL`, use smaller model, or increase cap (if within rules) |
| `ECONNREFUSED :7001` | MCP server not up | `docker compose logs mcp`; restart |
| n8n cannot save creds | Volume perms | Ensure `services/n8n/workflows` is writable |
| High latency p95 | Model too large | Switch to mini model for planning; cache prompts |

---

# 11) Security & Secrets

- All secrets via `.env` only; never in code or `workflow.json`.
- Rotate tokens if shared; least-privilege scopes for Slack/Google APIs.

---

# 12) Appendix – MCP How-To

**Adding a new tool to MCP**

1. Implement under `services/mcp/tools/<tool_name>.py` with `run(payload: dict) -> dict`.
2. Register in `server.py` and gate with `MCP_ALLOWED_TOOLS`.
3. Add env vars to `.env.example`.
4. Rebuild: `docker compose build mcp && docker compose up -d mcp`.

**Consuming MCP from Agents**

- API calls MCP over HTTP at `${MCP_HOST}:${MCP_PORT}` with JSON payloads.
- Use schemas to validate I/O. On error, retry with backoff.

---

## 13) Appendix – Example n8n Flow

- **Trigger**: Webhook → `/webhook/ingest`
- **Function**: Validate payload, attach `trace_id`
- **HTTP Node**: POST to `api:8000/invoke` with `{agent, input}`
- **IF**: On failure → Wait (backoff) → Retry up to 3
- **Slack** (optional): Post result to channel

---

## 14) License & Credits

- *Extra credit: link any OSS contributions (e.g., MCP servers).*
- List team roles & acknowledgments.

---

## 15) One-Pager (Non-Technical)

Include in slide deck and repo:

- Problem, why it matters, who benefits
- Short demo narrative (user → agents → tools → outcome)
- Guardrails (privacy, fairness) and civic impact