Gemini ADK + MCP — Kaggle Assignment Kit

A complete, minimal stack to: - Host a Gemini ADK agent with function tools (research + ML baseline). - Expose the same tools via an MCP server. - Drive the agent from a Kaggle notebook over HTTP (internet-enabled assignment).

---

0) Repo Layout

```
adk-mcp-kaggle/
├─ README.md
├─ .env.example
├─ requirements.txt
├─ docker/Dockerfile
├─ adk_app/
│  ├─ app.py
│  ├─ tools/
│  │  ├─ web_fetch.py
│  │  ├─ dataset_tools.py
│  │  ├─ cv_tools.py
│  │  ├─ baseline_tools.py
│  │  └─ report_tools.py
│  └─ policies/tool_allowlist.yaml
├─ mcp_server/
│  ├─ server.py
│  └─ schemas/
│     ├─ web_fetch.json
│     ├─ dataset.load_csv.json
│     ├─ cv.split.json
│     ├─ tabular.baseline.json
│     └─ report.md.json
└─ clients/
   └─ kaggle_client_cell.py
```

---

1) requirements.txt

```
fastapi
uvicorn
pydantic
httpx
python-dotenv
```

```
pandas
numpy
scikit-learn
lxml
beautifulsoup4
orjson
```

2) .env.example

```
GOOGLE_GENAI_API_KEY=YOUR_KEY
ADK_MODEL=gemini-2.5-flash
HOST=0.0.0.0
PORT=8080
```

3) ADK Agent (adk_app/app.py)

```python
import os
from fastapi import FastAPI
from fastapi.responses import JSONResponse
from pydantic import BaseModel
from typing import Dict, Any

# ---- ADK imports (placeholder for google.adk) ----
# Replace with actual ADK agent host when available in your environment
class Tool:
    def __init__(self, fn): self.fn = fn

class LlmAgent:
    def __init__(self, model: str, name: str, description: str, instruction:
str, tools: list):
        self.model = model
        self.name = name
        self.description = description
        self.instruction = instruction
        self.tools = {t.fn.__name__: t.fn for t in tools}
    def call_tool(self, name: str, **kwargs):
        return self.tools[name](**kwargs)

# ---- Tools ----
from tools.web_fetch import web_fetch
from tools.dataset_tools import dataset_list, dataset_load_csv
from tools.cv_tools import cv_split
from tools.baseline_tools import tabular_baseline
from tools.report_tools import report_md

agent = LlmAgent(
    model=os.getenv("ADK_MODEL", "gemini-2.5-flash"),
    name="kaggle_assignment_agent",
    description="Research + ML baseline agent with shared tools",
    instruction=(
        "Use web_fetch for external resources, then dataset_load_csv, cv_split,
tabular_baseline."
        " Keep outputs compact; return JSON-friendly results."
    ),
    tools=[Tool(web_fetch), Tool(dataset_list), Tool(dataset_load_csv),
Tool(cv_split), Tool(tabular_baseline), Tool(report_md)],
)

# ---- HTTP app ----
app = FastAPI()

class CallIn(BaseModel):
    tool: str
    args: Dict[str, Any] = {}

@app.post("/adk/call")
async def adk_call(inp: CallIn):
    try:
        out = agent.call_tool(inp.tool, **(inp.args or {}))
        return JSONResponse(out)
    except KeyError:
        return JSONResponse({"isError": True, "message": f"unknown tool
{inp.tool}"}, status_code=404)
    except Exception as e:
        return JSONResponse({"isError": True, "message": str(e)},
status_code=500)

@app.get("/health")
async def health():
    return {"status": "ok", "model": agent.model, "tools":
list(agent.tools.keys())}
```

4) Tools (adk_app/tools/*.py)

4.1 web_fetch.py

```
import httpx
from bs4 import BeautifulSoup

def web_fetch(url: str, timeout_sec: int = 20) -> dict:
    with httpx.Client(timeout=timeout_sec, follow_redirects=True) as s:
        r = s.get(url)
    ct = (r.headers.get("content-type") or "").lower()
    out = {"status": r.status_code, "headers": dict(r.headers), "url":
str(r.url)}
    if "html" in ct:
        soup = BeautifulSoup(r.text, "lxml")
        for t in soup(["script","style","noscript"]): t.decompose()
        md = "\n".join(h.get_text(" ", strip=True) for h in
soup.find_all(["h1","h2","h3","p","li"]))
        out.update({"title": soup.title.string if soup.title else None,
"markdown": md})
    else:
        out.update({"bytes": len(r.content)})
    return out
```

4.2 dataset_tools.py

```
import os, pandas as pd

BASE = "/kaggle/input"

def dataset_list() -> dict:
    items = []
    if os.path.isdir(BASE):
        for x in sorted(os.listdir(BASE)):
            p = os.path.join(BASE, x)
            if os.path.isdir(p): items.append({"name": x, "path": p})
    return {"datasets": items}

def dataset_load_csv(dataset: str, filename: str, nrows: int | None = None) ->
dict:
    path = os.path.join(BASE, dataset, filename)
    if not os.path.exists(path):
        return {"isError": True, "message": f"not found: {path}"}
    df = pd.read_csv(path, nrows=nrows)
    meta = {"rows": int(df.shape[0]), "cols": int(df.shape[1]), "columns":
```

```
df.columns.tolist()}
    cache = "/kaggle/working/agent/cache"
    os.makedirs(cache, exist_ok=True)
    feather_path = os.path.join(cache, f"{dataset}__{filename}.feather")
    try: df.to_feather(feather_path)
    except Exception: feather_path = ""
    return {"meta": meta, "head": df.head(5).to_dict(orient="list"),
"cache_feather": feather_path}
```

4.3 cv_tools.py

```
import numpy as np, json, os, pandas as pd
from sklearn.model_selection import StratifiedKFold, KFold

CACHE = "/kaggle/working/agent/cache"

def _load_cached_feather(path: str) -> pd.DataFrame:
    if path and os.path.exists(path):
        try: return pd.read_feather(path)
        except Exception: pass
    raise FileNotFoundError("cache not available; rerun dataset_load_csv")

def cv_split(cache_feather: str, target_col: str, n_splits: int = 5, stratified:
bool = True) -> dict:
    df = _load_cached_feather(cache_feather)
    if target_col not in df.columns: return {"isError": True, "message":
"target missing"}
    y = df[target_col].values
    idxs = np.arange(len(df))
    folds = []
    if stratified and df[target_col].nunique() <= 50 and df[target_col].dtype !=
float:
        it = StratifiedKFold(n_splits=n_splits, shuffle=True,
random_state=42).split(idxs, y)
    else:
        it = KFold(n_splits=n_splits, shuffle=True, random_state=42).split(idxs)
    for k,(tr,va) in enumerate(it):
        folds.append({"fold": k, "train_idx": tr.tolist(), "valid_idx":
va.tolist()})
    path = os.path.join(CACHE,
f"cv_{abs(hash((cache_feather,target_col,n_splits,stratified)))}.json")
    with open(path, "w") as f: json.dump({"target": target_col, "folds": folds},
f)
    return {"cv_path": path, "summary": [{"fold": f["fold"], "n_train":
len(f["train_idx"]), "n_valid": len(f["valid_idx"]) } for f in folds]}
```

4.4 baseline_tools.py

```
import os, json, math
import numpy as np, pandas as pd
from sklearn.preprocessing import StandardScaler
from sklearn.linear_model import LogisticRegression, Ridge
from sklearn.metrics import mean_squared_error, accuracy_score

CACHE = "/kaggle/working/agent/cache"
LOGS = "/kaggle/working/agent/logs"
REPORTS = "/kaggle/working/agent/reports"
for d in [LOGS, REPORTS]: os.makedirs(d, exist_ok=True)

def _load(df_path: str) -> pd.DataFrame:
    return pd.read_feather(df_path)

def _split_xy(df: pd.DataFrame, target: str):
    X =
df.drop(columns=[target]).select_dtypes(include=[np.number]).values.astype(np.float32)
    y = df[target].values
    return X, y

def tabular_baseline(cache_feather: str, cv_path: str, task: str = "auto") ->
dict:
    df = _load(cache_feather)
    with open(cv_path, "r") as f: cv = json.load(f)
    target = cv["target"]
    X_all, y_all = _split_xy(df, target)
    if task == "auto":
        if pd.api.types.is_numeric_dtype(df[target]) and df[target].nunique() >
max(20, int(len(df)*0.05)):
            task = "regression"
        else:
            task = "classification"
    scaler = StandardScaler(); X_all = scaler.fit_transform(X_all)
    oof = np.zeros(len(df), dtype=float); folds = []
    for f in cv["folds"]:
        tr, va = np.array(f["train_idx"]), np.array(f["valid_idx"])
        Xtr, Xva = X_all[tr], X_all[va]
        ytr, yva = y_all[tr], y_all[va]
        if task == "regression":
            m = Ridge(alpha=1.0, random_state=42).fit(Xtr, ytr)
            pred = m.predict(Xva); oof[va] = pred
            folds.append({"fold": f["fold"], "rmse":
float(math.sqrt(mean_squared_error(yva, pred)))})
        else:
            if not pd.api.types.is_integer_dtype(ytr):
                classes, ytr_enc = np.unique(ytr, return_inverse=True)
                yva_enc = np.searchsorted(classes, yva)
            else:
                classes, ytr_enc = np.unique(ytr, return_inverse=True); yva_enc
= yva
            m = LogisticRegression(max_iter=1000, random_state=42).fit(Xtr,
ytr_enc)
            pred = m.predict(Xva); oof[va] = pred
            folds.append({"fold": f["fold"], "accuracy":
float(accuracy_score(yva_enc, pred))})
    metrics_path = os.path.join(LOGS, "baseline_metrics.json")
    with open(metrics_path, "w") as f: json.dump({"task": task, "folds": folds,
"n": int(len(df))}, f, indent=2)
    oof_path = os.path.join(REPORTS, f"oof_{task}.csv");
pd.DataFrame({f"oof_{task}": oof}).to_csv(oof_path, index=False)
    return {"task": task, "fold_metrics": folds, "metrics_path": metrics_path,
"oof_path": oof_path}
```

4.5 report_tools.py

```
import os, json

REPORTS = "/kaggle/working/agent/reports"
os.makedirs(REPORTS, exist_ok=True)

def report_md(title: str, notes: list[str], metrics_json: str, links: list[dict]
| None = None) -> dict:
    try:
        with open(metrics_json,"r") as f: metrics = json.load(f)
    except Exception:
        metrics = {}
    body = [f"# {title}", "", "## Notes"] + [f"- {n}" for n in notes] + ["",
"## Metrics", "```json", json.dumps(metrics, indent=2), "```"]
    if links:
        body += ["", "## Sources"] + [f"- [{x.get('title', x['url'])}]
({x['url']})" for x in links]
    path = os.path.join(REPORTS, f"{title.lower().replace(' ','_')}.md")
    with open(path, "w", encoding="utf-8") as f: f.write("\n".join(body))
    return {"report_path": path}
```

5) MCP Server (mcp_server/server.py)

```python
from fastapi import FastAPI
from pydantic import BaseModel
from typing import Dict, Any
import json, importlib
from pathlib import Path

app = FastAPI()
SCHEMAS = Path(__file__).parent / "schemas"

# Map MCP tool names → ADK tool impls (same code)
TOOL_IMPLS = {
    "web_fetch": ("adk_app.tools.web_fetch", "web_fetch"),
    "dataset.load_csv": ("adk_app.tools.dataset_tools", "dataset_load_csv"),
    "cv.split": ("adk_app.tools.cv_tools", "cv_split"),
    "tabular.baseline": ("adk_app.tools.baseline_tools", "tabular_baseline"),
    "report.md": ("adk_app.tools.report_tools", "report_md"),
}

class MCPCall(BaseModel):
    name: str
    arguments: Dict[str, Any] = {}

@app.get("/mcp/tools")
def list_tools():
    tools = []
    for p in SCHEMAS.glob("*.json"):
        tools.append(json.loads(p.read_text()))
    return {"tools": tools}

@app.post("/mcp/call")
def call_tool(inp: MCPCall):
    if inp.name not in TOOL_IMPLS:
        return {"isError": True, "message": f"unknown tool {inp.name}"}
    mod_name, fn_name = TOOL_IMPLS[inp.name]
    mod = importlib.import_module(mod_name)
    fn = getattr(mod, fn_name)
    try:
        return fn(**(inp.arguments or {}))
    except Exception as e:
        return {"isError": True, "message": str(e)}
```

5.1 Example MCP schema (mcp_server/schemas/web_fetch.json)

```
{
  "name": "web_fetch",
  "title": "Fetch and normalize a web page",
  "description": "GET a URL; return markdown or metadata",
  "inputSchema": {
    "type": "object",
    "properties": {"url": {"type": "string"}, "timeout_sec": {"type":
"integer", "default": 20}},
    "required": ["url"]
  },
  "outputSchema": {
    "type": "object",
    "properties": {
      "status": {"type": "integer"},
      "title": {"type": ["string", "null"]},
      "markdown": {"type": ["string", "null"]},
      "headers": {"type": "object"},
      "url": {"type": "string"}
    },
    "required": ["status", "url"]
  }
}
```

(Replicate with appropriate schemas for the other tools.)

---

6) Run locally

```
python -m venv .venv && source .venv/bin/activate
pip install -r requirements.txt
cp .env.example .env  # add GOOGLE_GENAI_API_KEY
uvicorn adk_app.app:app --host 0.0.0.0 --port 8080 &
uvicorn mcp_server.server:app --host 0.0.0.0 --port 8081 &
```

- ADK endpoint: `POST /adk/call` with `{tool:"web_fetch", args:{...}}` - MCP tool list: `GET /mcp/tools`, call: `POST /mcp/call` with `{name:"web_fetch", arguments:{...}}`

---

7) Kaggle client (clients/kaggle_client_cell.py)

```python
# Drop this cell into Kaggle. Set base URLs of your running services.
import os, json, httpx
ADK_BASE = os.environ.get("ADK_BASE", "https://<your-host>:8080")
MCP_BASE = os.environ.get("MCP_BASE", "https://<your-host>:8081")

# 1) Web fetch via ADK
with httpx.Client(timeout=20) as s:
    r = s.post(f"{ADK_BASE}/adk/call", json={"tool":"web_fetch","args":
{"url":"https://archive.ics.uci.edu/ml/datasets/Wine+Quality"}})
    fetched = r.json()
print(fetched.get("title"), len(fetched.get("markdown","")))

# 2) Load Kaggle dataset and run baseline fully via ADK
with httpx.Client(timeout=20) as s:
    load = s.post(f"{ADK_BASE}/adk/call",
json={"tool":"dataset_load_csv","args":
{"dataset":"titanic","filename":"train.csv"}}).json()
    cv   = s.post(f"{ADK_BASE}/adk/call", json={"tool":"cv_split","args":
{"cache_feather":load["cache_feather"],"target_col":"Survived","n_splits":
5,"stratified":True}}).json()
    base = s.post(f"{ADK_BASE}/adk/call",
json={"tool":"tabular_baseline","args":
{"cache_feather":load["cache_feather"],"cv_path":cv["cv_path"],"task":"classification"}}).json()
print(base)

# 3) Same calls via MCP
with httpx.Client(timeout=20) as s:
    r = s.get(f"{MCP_BASE}/mcp/tools").json(); print([t["name"] for t in
r["tools"]])
    mf = s.post(f"{MCP_BASE}/mcp/call", json={"name":"web_fetch","arguments":
{"url":"https://example.com"}}).json()
print(mf["status"])
```

---

8) Notes

- The ADK stand-in here routes function tools; when running in Google's ADK environment, swap the shim classes with actual imports (e.g., `from google.adk.agents import LlmAgent, Tool`).
- The MCP server exposes identical capabilities with JSON schemas for discoverability and validation.
- The Kaggle client uses plain HTTP; keep your host public or tunnel (where allowed) for the assignment.
- For grading, submit the notebook plus generated artifacts under `/kaggle/working/agent/`.