Gideon — Full Starter Scaffold (All files)

This document contains a ready-to-copy **starter scaffold** for the Gideon project described earlier. It includes the repository layout and **every file's full content** you need to get the MVP running locally (MVP Gideon core, sample agent, trading agent scaffold, infra, manifest schema, agent SDK, and README). Copy each file into your project with the same path and names.

Repo layout

```
/gideon
  /gideon-core
    main.py
    registry.py
    planner.py
    models/llm.py
    requirements.txt
    .env.example
  /agents
    /sample_agent
      agent.py
      manifest.json
      requirements.txt
    /trading_agent
      agent.py
      manifest.json
      requirements.txt
      README.md
      /backtest
        example_data.csv
        backtest.py
    sdk.py
  /infra
    docker-compose.yml
  /docs
    manifest.schema.json
    security.md
  README.md
  .gitignore
```

Note: This scaffold is opinionated for clarity and ease of getting started. It uses Python + FastAPI services, Docker Compose with Qdrant and Redis, and simple HTTP-based plugin registration.

File contents — copy each file into your repo

`.gitignore`

```
# Python
__pycache__/
*.pyc
*.pyo
*.pyd
.env
venv/
.venv/

# Docker
docker-compose.override.yml

# Editor
.vscode/
.idea/

# Qdrant/DB
qdrant-data/

# Logs
*.log

# Mac
.DS_Store
```

`README.md`

```
# Gideon — Starter Scaffold

This repository is a starter scaffold for *Gideon*, a local modular AI
orchestrator. It provides an MVP: a Gideon core (FastAPI), a sample agent
```

~~~
(echo), a trading agent scaffold (simulation/backtest endpoints), infra (Qdrant,
Redis), and utilities (agent SDK, manifest schema).

## Quick start (dev)
1. Install Docker and Docker Compose.
2. Start infra services (Qdrant + Redis):
```bash
cd infra
docker compose up -d
~~~

3. Create & activate a Python venv at repo root and install dependencies for `gideon-core` and the agents you want to run.

Example (mac / linux):

```
python3.11 -m venv .venv
source .venv/bin/activate
pip install -r gideon-core/requirements.txt
pip install -r agents/sample_agent/requirements.txt
pip install -r agents/trading_agent/requirements.txt
```

1. Run services in separate terminals:

```
# Terminal 1 — Gideon core
cd gideon-core
uvicorn main:app --reload --port 8000

# Terminal 2 — sample agent
cd agents/sample_agent
uvicorn agent:app --reload --port 8102

# Terminal 3 — trading agent (simulation scaffold)
cd agents/trading_agent
uvicorn agent:app --reload --port 8202
```

2. Register agents with Gideon Core and test:

```
curl -X POST "http://localhost:8000/register" -H "Content-Type:
application/json" -d @agents/sample_agent/manifest.json
curl -X POST "http://localhost:8000/register" -H "Content-Type:
application/json" -d @agents/trading_agent/manifest.json
```

```
curl -X POST "http://localhost:8000/ask" -H "Content-Type: application/
json" -d '{"text":"hello Gideon"}'
```

What's included

- `gideon-core`: orchestrator, registry, planner stub, LLM wrapper (cloud/local switch).
- `agents`: sample echo agent, trading agent scaffold (simulate/backtest endpoints) and `sdk.py` to help agent registration.
- `infra`: docker-compose with Qdrant and Redis.
- `docs/manifest.schema.json`: JSON Schema for agent manifests.

Next steps

- Replace LLM wrapper to use your preferred provider or local model.
- Implement more advanced planner logic in `planner.py`.
- Harden security and implement persistent token-based registration.

~~~
---

## `/infra/docker-compose.yml`
```yaml
version: "3.8"
services:
  qdrant:
    image: qdrant/qdrant:latest
    restart: unless-stopped
    ports:
      - "6333:6333"
    volumes:
      - qdrant-data:/qdrant/storage

  redis:
    image: redis:7
    restart: unless-stopped
    ports:
      - "6379:6379"

volumes:
  qdrant-data:
~~~

`/docs/manifest.schema.json`

```
{
  "$schema": "http://json-schema.org/draft-07/schema#",
  "title": "Agent Manifest",
  "type": "object",
  "properties": {
    "name": {"type": "string"},
    "version": {"type": "string"},
    "capabilities": {"type": "array", "items": {"type": "string"}},
    "requires_permissions": {"type": "array", "items": {"type": "string"}},
    "endpoint": {"type": "string", "format": "uri"},
    "healthcheck": {"type": "string"}
  },
  "required": ["name","version","capabilities","endpoint"]
}
```

`/docs/security.md`

```
# Security notes (starter)

This file summarizes starter security practices you should adopt before enabling
dangerous capabilities (like trading execution).

- **Secrets**: do not store API keys in source control. Use Vault or environment
variables and encrypted secrets for CI.
- **Agent permissions**: each agent must declare `requires_permissions`. Gideon
must enforce RBAC and require human confirmation for privileged actions.
- **Sandboxing**: run untrusted agents in containers with limited capabilities
and network access.
- **Auditing**: log all actions and keep append-only audit logs for at least 90
days.
- **Trading**: `execute` endpoints must require multi-factor confirmation and be
disabled by default.
```

`/gideon-core/.env.example`

```
# Gideon Core configuration example
GIDEON_HOST=0.0.0.0
GIDEON_PORT=8000
```

```
QDRANT_URL=http://localhost:6333
REDIS_URL=redis://localhost:6379
OPENAI_API_KEY=
```

`/gideon-core/requirements.txt`

```
fastapi
uvicorn[standard]
requests
pydantic
python-dotenv
qdrant-client
sentence-transformers
redis
```

`/gideon-core/main.py`

```python
# gideon-core/main.py
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
from registry import AgentRegistry
from planner import Planner
import os
import uuid

app = FastAPI(title="Gideon Core - MVP")
registry = AgentRegistry()
planner = Planner(registry)

class Query(BaseModel):
    text: str
    session_id: str | None = None

@app.post("/register")
def register_agent(manifest: dict):
    """Register an agent manifest (simple POST registration)."""
    try:
        name = manifest.get("name")
        if not name:
            raise HTTPException(status_code=400, detail="missing name")
        registry.register(manifest)
        return {"status": "ok", "registered": name}
    except Exception as e:
        raise HTTPException(status_code=400, detail=str(e))

@app.get("/agents")
def list_agents():
    return registry.list_all()

@app.post("/ask")
def ask(q: Query):
    """Ask Gideon a question. Planner will decide whether to call an agent or
respond.
    Returns either an agent result or LLM-based answer (via planner).
    """
    session_id = q.session_id or str(uuid.uuid4())
    # Planner returns a dict containing either 'reply' or 'call_agent'
    result = planner.handle_text(q.text, session_id=session_id)
    return result

@app.get("/health")
def health():
    return {"status": "ok", "agents": len(registry.list_all())}
```

/gideon-core/registry.py

```
# gideon-core/registry.py
"""
Very small registry implementation. Replaces with DB later.
"""
from typing import Dict

class AgentRegistry:
    def __init__(self):
        # store manifest keyed by name
        self._agents: Dict[str, dict] = {}

    def register(self, manifest: dict):
        name = manifest.get("name")
        if not name:
            raise ValueError("manifest must include a name")
        self._agents[name] = manifest

    def get(self, name: str) -> dict | None:
        return self._agents.get(name)
    def list_all(self):
        return list(self._agents.values())

    def pick_for_intent(self, intent: str) -> dict | None:

# VERY naive - pick the first agent that lists a capability matching the intent
token
        for m in self._agents.values():
            caps = [c.lower() for c in m.get("capabilities", [])]
            if intent.lower() in caps:
                return m
        # fallback: return first agent
        return next(iter(self._agents.values()), None)
```

/gideon-core/planner.py

```
# gideon-core/planner.py
"""
Planner decides when to call an agent vs. to ask the LLM to respond.
This MVP planner is deliberately simple and rule-based. Replace with an LLM-
based planner later.
"""
import requests
from models.llm import LLM

class Planner:
    def __init__(self, registry):
        self.registry = registry
        # LLM wrapper (can be cloud or local)
        self.llm = LLM()

    def handle_text(self, text: str, session_id: str):
        # naive intent detection: look for keywords
        lowered = text.lower()
        if any(k in lowered for k in ["trade", "buy", "sell", "position",
"simulate"]):
            # ask registry for trading agent
            agent = self.registry.pick_for_intent("trading")
            if not agent:
                return {"error": "no trading agent registered"}
            # call agent's /simulate endpoint if the user asked to simulate
            endpoint = agent["endpoint"].rstrip("/")
            # pick simulate vs execute by keyword
            if "simulate" in lowered or "backtest" in lowered:
                url = endpoint + "/simulate"
                payload = {"text": text, "session_id": session_id}
                try:
                    resp = requests.post(url, json=payload, timeout=20).json()
                    return {"from_agent": agent["name"], "result": resp}
                except Exception as e:
                    return {"error": f"agent call failed: {e}"}
            else:
                # For non-simulate trade intents, ask LLM to produce a
structured plan
                prompt = self._build_prompt(text, session_id)
                ans = self.llm.generate(prompt)
                return {"from_llm": True, "reply": ans}
        else:
            # no special intent — use LLM to answer
            prompt = self._build_prompt(text, session_id)
            ans = self.llm.generate(prompt)
            return {"from_llm": True, "reply": ans}

    def _build_prompt(self, text, session_id):
        system = "You are Gideon, a local assistant. Be helpful, concise, and
safe."
        prompt = f"{system}\n\nSession: {session_id}\nUser: {text}\n\nReply:"
        return prompt
```

/gideon-core/models/llm.py

```
# gideon-core/models/llm.py
"""
A tiny LLM wrapper. Default uses OpenAI if OPENAI_API_KEY env var present;
otherwise uses a mock/simplified response.
Replace with local model inference code when you switch to a local model.
"""
import os

OPENAI_KEY = os.getenv("OPENAI_API_KEY")

class LLM:
    def __init__(self):
        self.key = OPENAI_KEY
        if self.key:
            try:
                import openai
                self.client = openai
                self.client.api_key = self.key
            except Exception:
                self.client = None
        else:
            self.client = None

    def generate(self, prompt: str, max_tokens: int = 256) -> str:
        if self.client:
            try:
                resp = self.client.ChatCompletion.create(
                    model="gpt-3.5-turbo",
                    messages=[{"role":"system","content":"You are Gideon."},
{"role":"user","content":prompt}],
                    max_tokens=max_tokens,
                    temperature=0.2,
                )
                return resp.choices[0].message.content.strip()
            except Exception as e:
                return f"LLM call failed: {e}"
        # fallback: simple echo / mock
        return f"(mock reply) I understood: {prompt[:200]}"
```

`/agents/sdk.py`

```
# agents/sdk.py
"""
Simple agent SDK to register an agent with Gideon Core at startup.
Usage: call register_with_gideon(gideon_url, manifest)
"""
import os
import requests
import time


def register_with_gideon(gideon_url: str, manifest: dict, retries: int = 3,
delay: float = 1.0):
    url = gideon_url.rstrip("/") + "/register"
    for i in range(retries):
        try:
            resp = requests.post(url, json=manifest, timeout=5)
            resp.raise_for_status()
            print("registered with gideon:", resp.json())
            return True
        except Exception as e:
            print(f"register attempt failed: {e}")
            time.sleep(delay)
    return False

if __name__ == "__main__":
    # quick smoke test when running directly
    gideon = os.getenv("GIDEON_URL", "http://localhost:8000")
    import json
    sample_manifest = {
        "name": "sample_agent",
        "version": "0.0.1",
        "capabilities": ["echo"],
        "requires_permissions": [],
        "endpoint": "http://localhost:8102",
        "healthcheck": "/health"
    }
    register_with_gideon(gideon, sample_manifest)
```

`/agents/sample_agent/requirements.txt`

```
fastapi
uvicorn[standard]
pydantic
requests
```

`/agents/sample_agent/manifest.json`

```
{
  "name": "sample_agent",
  "version": "0.0.1",
  "capabilities": ["echo"],
  "requires_permissions": [],
  "endpoint": "http://localhost:8102",
  "healthcheck": "/health"
}
```

/agents/sample_agent/agent.py

```python
# agents/sample_agent/agent.py
from fastapi import FastAPI
from pydantic import BaseModel
import uvicorn
import os

app = FastAPI(title="Sample Agent")

class HandleReq(BaseModel):
    text: str
    session_id: str | None = None

@app.get("/health")
def health():
    return {"status": "ok"}

@app.post("/handle")
def handle(req: HandleReq):
    # naive processing - echos the input
    return {"agent": "sample_agent", "reply": f"Echo: {req.text}"}

if __name__ == "__main__":
    uvicorn.run("agent:app", port=8102, reload=True)
```

/agents/trading_agent/requirements.txt

```
fastapi
uvicorn[standard]
pydantic
requests
pandas
numpy
```

/agents/trading_agent/manifest.json

```json
{
  "name": "trading_agent",
  "version": "0.1.0",
  "capabilities": ["trading", "simulate", "backtest"],
  "requires_permissions": ["broker:trade"],
  "endpoint": "http://localhost:8202",
  "healthcheck": "/health"
}
```

`/agents/trading_agent/agent.py`

```
# agents/trading_agent/agent.py
from fastapi import FastAPI, HTTPException
from pydantic import BaseModel
import uvicorn
import pandas as pd
import os

app = FastAPI(title="Trading Agent - Simulation Scaffold")

class SimReq(BaseModel):
    text: str
    session_id: str | None = None
    params: dict | None = None

class ExecReq(BaseModel):
    strategy: str
    symbol: str
    size: float
    session_id: str | None = None
    auth_token: str | None = None
    mfa_code: str | None = None

@app.get("/health")
def health():
    return {"status": "ok"}

@app.post("/simulate")
def simulate(req: SimReq):
    # very naive: parse simple instructions from text and run the example
backtest
    # Returns a mock backtest summary
    # In a real agent, you'd parse params and run backtest/backtrader
    try:
        # use bundled example CSV in /backtest/example_data.csv
        csv = os.path.join(os.path.dirname(__file__), "backtest",
"example_data.csv")
```

```
        df = pd.read_csv(csv)
        # perform an extremely naive moving-average crossover backtest for
demonstration
        df['ma_fast'] = df['close'].rolling(window=5).mean()
        df['ma_slow'] = df['close'].rolling(window=20).mean()
        df = df.dropna()
        position = 0
        cash = 100000.0
        shares = 0
        for _, row in df.iterrows():
            if row['ma_fast'] > row['ma_slow'] and position == 0:
                # buy
                shares = cash // row['close']
                cost = shares * row['close']
                cash -= cost
                position = 1
            elif row['ma_fast'] < row['ma_slow'] and position == 1:
                cash += shares * row['close']
                shares = 0
                position = 0
        # compute final value
        final_value = cash + shares * df.iloc[-1]['close']
        return {"sim": True, "start_cash": 100000.0, "final_value": final_value,
"notes": "naive MA crossover demo"}
    except Exception as e:
        raise HTTPException(status_code=500, detail=str(e))

@app.post("/execute")
def execute(req: ExecReq):

# DO NOT execute live trades in this scaffold. This endpoint requires proper
auth and MFA in production.
    return {"executed": False, "reason": "execute disabled in scaffold.
Implement broker adapter and MFA before enabling."}

@app.get("/positions")
def positions():
    return {"positions": []}

if __name__ == "__main__":
    uvicorn.run("agent:app", port=8202, reload=True)
```

`/agents/trading_agent/README.md`

```
# Trading Agent (Simulation Scaffold)

This agent provides demonstration endpoints for `simulate` and `execute`
(execute disabled by default). It includes a tiny naive backtest implemented in
`/backtest/backtest.py` using the bundled `example_data.csv`.

**Important:** Do not enable live execution without adding proper broker
adapters, secure secrets storage, and MFA.
```

`/agents/trading_agent/backtest/example_data.csv`

```
# date,open,high,low,close,volume
date,open,high,low,close,volume
2023-01-01,100,101,99,100,1000
2023-01-02,100,102,99,101,1100
2023-01-03,101,103,100,102,1200
2023-01-04,102,104,101,103,1300
2023-01-05,103,105,102,104,1400
2023-01-06,104,106,103,105,1500
2023-01-07,105,107,104,106,1600
2023-01-08,106,108,105,107,1700
2023-01-09,107,109,106,108,1800
2023-01-10,108,110,107,109,1900
2023-01-11,109,111,108,110,2000
2023-01-12,110,112,109,111,2100
2023-01-13,111,113,110,112,2200
2023-01-14,112,114,111,113,2300
2023-01-15,113,115,112,114,2400
2023-01-16,114,116,113,115,2500
2023-01-17,115,117,114,116,2600
2023-01-18,116,118,115,117,2700
2023-01-19,117,119,116,118,2800
2023-01-20,118,120,117,119,2900
```

Usage examples

After starting the three services (gideon-core, sample_agent, trading_agent), register agents with Gideon:

```
curl -X POST "http://localhost:8000/register" -H "Content-Type: application/
json" -d @agents/sample_agent/manifest.json
curl -X POST "http://localhost:8000/register" -H "Content-Type: application/
json" -d @agents/trading_agent/manifest.json
```

Ask Gideon to echo:

```
curl -X POST "http://localhost:8000/ask" -H "Content-Type: application/json" -d
'{"text":"hello Gideon"}'
```

Ask Gideon to simulate trading/backtest:

```
curl -X POST "http://localhost:8000/ask" -H "Content-Type: application/json" -d
'{"text":"simulate moving average backtest for AAPL"}'
```

Next steps (dev suggestions)

1. Replace `models/llm.py` with your preferred cloud or local LLM integration.
2. Implement a persistent registry (SQLite or Postgres) and token-based secure registration for agents.
3. Implement vector DB memory & RAG flow using Qdrant (`qdrant-client`) and embedder (sentence-transformers).
4. Add speech-service, STT/TTS pipeline (Whisper/Coqui) and integrate voice turn-taking with the planner.
5. Harden security (Vault, RBAC), sandbox agents, add observability (Prometheus, Jaeger), and CI/CD.

If you want, I can now: - produce a single `docker-compose.yml` that also containerizes the `gideon-core` and the two agents, or - generate a more feature-complete `models/llm.py` that uses a local quantized model, or - produce unit test examples and CI config.

Tell me which next artifact you want and I'll add it into the scaffold.