Comprehensive, In-Depth Implementation Plan for a Python-Based Dual-Protocol Orchestrator

This document provides step-by-step, highly detailed guidance for the next engineer ("Agent") who will port the Go-based MCP aggregator to Python, integrate A2A support, embed GPT-OSS reasoning via Hugging Face, and deliver a production-ready CLI and library—all managed via `uv`.

Table of Contents

1. Project Initialization

1.1. **Repository Creation**

- Create GitHub repo `yourorg/orchestrator`
- Enable Issues, Projects, and Wiki

1.2. **Default Branch Protection**

- Require pull-request reviews
- Enforce status checks (lint, tests)

1.3. **Collaborator Access**

- Grant write access to core team members
- Invite CI bots and project management integrations

2. Environment Setup with `uv`

2.1. **Install** `uv`

```
pip install uv
```

2.2. **Initialize Project**

```
cd orchestrator
uv init
```

2.3. **Define Dependencies**

Edit `uv.toml`:

```
[tool.uv]
python = ">=3.11"
packages = [
  "typer>=0.9",
  "pydantic>=2.0",
  "httpx>=0.24",
  "aiohttp>=3.8",
  "openai>=0.27",
  "huggingface-hub>=0.13",
  "authlib>=1.2",
  "pytest-asyncio>=0.21",
  "sseclient-py>=1.7"
]
```

2.4. **Lock Dependencies**

```
uv lock
```

2.5. **Activate Environment**

```
uv shell
```

3. Repository Structure & Packaging

```
orchestrator/
├─ orchestrator/               # Core library
│  ├─ drivers/
│  │  ├─ mcp.py                # MCPDriver implementation
```

```
│   │   └─ a2a.py                # A2ADriver implementation
│   ├─ config.py                 # Pydantic settings models
│   ├─ orchestrator.py           # Central orchestration logic
│   ├─ query_analyzer.py         # GPT-OSS integration
│   ├─ server_selector.py        # Protocol & agent selection logic
│   └─ workflows.py              # Workflow engine & state management
├─ cli/
│   └─ ts.py                     # Typer-based CLI entrypoint
├─ examples/
│   ├─ basic_usage.sh
│   └─ advanced_workflow.yaml
├─ tests/
│   ├─ test_mcp_driver.py
│   ├─ test_a2a_driver.py
│   ├─ test_orchestrator.py
│   └─ test_cli.py
├─ uv.toml                       # uv config
├─ uv.lock                       # locked dependencies
├─ Dockerfile                    # containerization
├─ pyproject.toml                # metadata (optional)
└─ README.md
```

3.1. **Package Names & Imports**

- Core library import path: `import orchestrator`
- CLI module: `from cli.ts import app`

3.2. **Versioning**

- Use semantic versioning in `pyproject.toml` or as a constant in `orchestrator/__init__.py`

4. Phase 1: Foundation & Porting (Weeks 1–3)

4.1. Port MCP Aggregator to Python

4.1.1. **Subprocess & STDIO Transport**

- In `drivers/mcp.py`, implement `async def initialize(self):`, `async def list_tools(self)`, `async def call_tool(self, name, params)`.
- Spawn with `await asyncio.create_subprocess_exec()` using server command and args.
- Pipe stdout → parse JSON, stderr → log.

4.1.2. **SSE Transport (Optional)**

- Use `aiohttp.ClientSession().get(..., timeout=None)` to consume SSE from `http://server/sse`.
- Parse events, map to `call_tool` results.

4.1.3. **Tool Discovery & Prefixing**

- Mirror combine-mcp's `discoverTools` logic: list tools, sanitize names, prefix with server name.

- Implement filtering by allowed tools in config.

4.2. Build A2A Driver

4.2.1. **HTTP + JSON-RPC**

- In `drivers/a2a.py`, create `A2ADriver` with `async def discover_agents(self):` calling `/discovery` endpoint.
- Implement `async def call_agent(self, agent_id, method, params)` sending JSON-RPC requests via `httpx.AsyncClient`.

4.2.2. **Authentication**

- Support bearer tokens: pass `Authorization: Bearer <token>` header.
- Accept OAuth2 client credentials flow if agent endpoints require it.

4.2.3. **Event Subscriptions (Optional)**

- Support WebSocket or pub/sub for asynchronous callbacks using `websockets` or `httpx` streaming.

4.3. Configuration Models

4.3.1. **Pydantic Settings**

- In `config.py`, define `class MCPServer(BaseModel)`, `class A2AEndpoint(BaseModel)`, `class Settings(BaseSettings)` reading from `~/.ts/config.yaml` or env vars.

4.3.2. **Config Loading**

```
settings = Settings(_env_file="~/.ts/config.env", config_file="~/.ts/config.yaml")
```

4.4. CLI Scaffolding

4.4.1. **Typer App**

- In `cli/ts.py`, create `app = Typer()`.
- Decorate functions:
- `@app.command("q") → async def query(...)`
- `@app.command("orchestrate")`
- `@app.command("discover")`
- `@app.command("workflow")`

4.4.2. **Flag Definitions**

- Use `Option(False, "-f", "--faves")`, `Option(False, "-u", "--useful")`, etc.
- Add `--words` as `Option(None, "-w", "--words")`.

4.4.3. **Alias Support**

- In entrypoint script (`__main__.py`), map `ts` to `app()`.

4.5. Basic Orchestration Logic

4.5.1. **Orchestrator Class**

- In `orchestrator/orchestrator.py`, implement `async def handle_query(self, text, modes, words):`
- Call `QueryAnalyzer` to get `plan = {protocols: [...], agents: [...]}`
- Dispatch calls: `await self.run_protocol(protocol, agents)`

4.5.2. **Protocol Driver Manager**

- Factory function choosing between `MCPDriver` and `A2ADriver` based on plan.
- Enforce single-instance logic for duplicate agents/servers.

5. Phase 2: Intelligence Integration (Weeks 4–8)

5.1. Hugging Face & GPT-OSS Setup

5.1.1. **HF Pro Account**

- Log into HF, navigate to **Settings** → **Access Tokens** → generate token → set `HF_API_KEY`.

5.1.2. **Model Selection**

- Use `openai/gpt-oss-20B` for `faves`, `openai/gpt-oss-120B` for `essential`.

5.2. QueryAnalyzer

5.2.1. **Prompt Engineering**

- Create structured prompt with sections:

1. "Available MCP servers: ..."
2. "Available A2A agents: ..."
3. "User request: ..."
4. "Mode flags: ..."
5. "Words filter: ..."

5.2.2. **API Call**

```
from huggingface_hub import InferenceClient
client = InferenceClient(token=settings.hf_api_key)
response = client.text_generation(prompt, model=settings.gpt_oss_model, max_new_tokens=ma
plan = json.loads(response.generated_text)
```

5.2.3. **Response Validation**

- Validate `plan` against Pydantic schema: `class Plan(BaseModel)` with `protocols: List[str]`, `agents: List[str]`.

5.3. ServerSelector

5.3.1. **Mapping**

- Map `plan.protocols` to driver classes.
- Map `plan.agents` to instances configured in `settings`.

6. Phase 3: Advanced Orchestration & Persistence (Weeks 9–14)

6.1. Workflow Engine

6.1.1. **State Model**

- Use SQLite (via `aiosqlite`) or Redis for:
- Workflow ID
- Step definitions
- Status (pending, running, succeeded, failed)

6.1.2. **Execution Engine**

- Represent workflows as DAGs: nodes = driver calls, edges = dependencies.
- Use `networkx` or custom scheduler to run independent steps in parallel.

6.1.3. **Failure Handling**

- Circuit breaker: track consecutive failures per agent.
- Configurable retry policies: `{"retries":3, "delay":5}`.

6.2. Security & Authentication

6.2.1. **OAuth2 for A2A**

- In `a2a.py`, implement `Authlib` OAuth2 client credentials flow.
- Refresh tokens automatically when expired.

6.2.2. **Encryption**

- Store HF and A2A tokens in OS keyring via `keyring` library.
- Encrypt local config file with `python-gnupg` if needed.

6.3. Multi-Modal Agent Support

6.3.1. **WebSockets**

- If A2A agents support streaming, use `websockets` for bi-directional channels.

6.3.2. **Media Handling**

- Integrate `aiortc` for audio/video if required by future agent use cases.

6.4. Cost & Usage Monitoring

6.4.1. **Token Tracking**

- Wrap HF calls to subtract `usage.total_tokens` from a running budget.
- Log consumption per query and enforce soft limit at 80% of $50 credit.

7. Phase 4: Production Readiness & Ecosystem (Weeks 15–18)

7.1. Testing

- **Unit Tests**: Mock drivers, test individual methods.
- **Integration Tests**: Spin up dummy MCP server via `subprocess` of `combine-mcp` binary, mock A2A HTTP server with `pytest-aiohttp`.
- **E2E Tests**: Example scripts under `examples/` invoking `ts q …`.

7.2. CI/CD

- **GitHub Actions**:
  1. `uv install`
  2. `uv run pytest`
  3. `uv run flake8`
  4. Build & push Docker image on `main`

7.3. Documentation

- README.md: Overview, Quickstart, CLI reference, Examples.
- **CLI Help**: Auto-generated via Typer.
- **Detailed Docs**: Use MkDocs or Sphinx for deeper guides.

7.4. Packaging & Release

- **PyPI**: `uv publish`
- **Docker**: `docker build -t yourorg/orchestrator:latest .`
- **GitHub Release**: Tag version, attach wheels and Docker tags.

7.5. Community Engagement

- Submit a blog post on HF and MCP/A2A communities.
- Demo at relevant meetups or conferences.
- Open RFCs for protocol enhancements based on real-world feedback.

8. Hugging Face & GPT-OSS Configuration

8.1. **HF Pro Account**

- Activate Inference API on your `huggingface.co` profile.
- Request elevated rate limits if needed.

8.2. **API Key Setup**

- Store `HF_API_KEY` in environment or keyring.
- Validate with a quick test:

```
uv run python - <<EOF
from huggingface_hub import InferenceClient
print(InferenceClient(token="...").model_info("openai/gpt-oss-20B"))
EOF
```

8.3. **GPT-OSS Credit Allocation**

- Dev allocations stored in code with soft caps:
  - Phase 2: 40% credits
  - Phase 3: 40% credits
  - Phase 4: 20% credits

9. Resource & Cost Management

- **Token Budgeting**: Track and log token use.
- **Model Tiering**:
  - `faves`: `gpt-oss-20B`
  - `useful`: `gpt-oss-20B` or `gpt-oss-120B` (if medium)
  - `essential`: `gpt-oss-120B`
- **Caching**: Memoize repeated `QueryAnalyzer` calls for identical prompts.

10. Security & Compliance

- **Secrets Management**: `keyring` + `Authlib` for tokens.
- **Transport Security**: Enforce HTTPS/TLS for all A2A endpoints.
- **Input Validation**: Pydantic schemas for all RPC payloads.
- **Logging & Audit**: Structured logs with correlation IDs.

11. Testing, CI/CD, and Quality Assurance

- **Static Analysis**: `flake8`, `mypy`.
- **Test Coverage**: Aim ≥90% coverage.
- **Automated Releases**: Semantic versioning via GH Actions.

12. Documentation & Community Engagement

- **Getting Started Guide**
- **Deep Dive Tutorials**: Porting logic, creating new drivers
- **API Reference**: Auto-generated from docstrings
- **Community Feedback Loop**: GitHub Discussions, Slack/Discord channel

13. Success Metrics & Monitoring

- **Performance**: <2 s simple queries, <10 s complex flows
- **Reliability**: ≥99.5% uptime
- **Adoption**: ≥100 stars, 20 forks in 3 months
- **Cost Efficiency**: ≤$50 credits consumption for MVP
- **Community**: ≥10 external contributors

This exhaustive, step-by-step plan equips the next engineer with every detail—from environment setup and porting strategy to orchestration engine design, security, testing, and community engagement—ensuring a smooth, efficient implementation of a Python-based dual-protocol orchestrator.