Microservice Canonicalization & Onboarding Guide

This document defines the **canonical microservice pattern**, the **naming conventions**, and the **transformation workflow** used to migrate existing services into a unified structure that the AppFactory and future agents can reliably consume. It also contains the **onboarding prompt** for any GPT/Gemini agent assisting in the transformation.

---

1. Purpose

The goal is to unify the microservice library so that: - Each service exposes a **predictable, machine-readable public surface**. - The AppFactory can introspect services, wire them together, and import them cleanly. - Transformation agents can convert arbitrary Python logic into canonical microservices using the TokenizingPATCHER. - New microservices can be generated by agents with minimal instructions and reliably integrated.

The strategy: 1. Define **canonical boilerplate** (the schema). 2. Enforce **naming conventions**. 3. Provide onboarding instructions for **transformation agents**.

---

2. Folder & Naming Conventions

Each microservice lives inside its own folder:

```
_<ServiceName>MS/
```

Examples: - `_SearchEngineMS/` - `_WebScraperMS/` - `_PromptVaultMS/`

Inside this folder, the canonical entry module is:

```
app.py
```

And inside `app.py`, the main class is:

```
class <ServiceName>MS:
```

This naming pattern is rigid and must be preserved.

2.1 Internal file naming

Additional internal files may exist, with normalized names:

```
helpers.py      # internal utility functions
storage.py      # persistence logic
adapters.py     # external API/db adapters
ui.py           # UI helpers
config.json     # optional config
state.json      # optional runtime state
```

The rule:

> **Only `app.py` defines the public surface. Everything else is internal.**

---

3. Canonical Microservice Boilerplate

The following pattern defines what a transformed microservice must look like.

```python
from typing import Any, Dict, List, Optional
from microservice_std_lib import service_endpoint


class {ServiceName}MS:
    """
    One-sentence summary of what this service does.
    Optionally: a longer explanation.
    """

    def __init__(self, config: Optional[Dict[str, Any]] = None):
        self.config = config or {}
        # migrate any existing init/setup logic here

    # === Public Endpoints ===
    @service_endpoint(
        inputs={
            # "arg_name": type,
        },
        outputs={
            # "result": "List[Dict]" or similar
        },
        description="Present-tense description of the endpoint.",
        tags=["search", "ingest", "kb", "ui", "safety", ...],
```

```
    )
    def endpoint_method(self, ...):
        # original logic (preserved)
        ...
        return ...

    # === Internal helpers (no decorator) ===
    def _helper_method(self, ...):
        ...


if __name__ == "__main__":
    svc = {ServiceName}MS()
    print("Service ready:", svc)
    # Optional: tiny example call
```

3.1 Public Endpoint Rule

A method becomes a public `@service_endpoint` when:

> **Another microservice or an AppFactory-built app could realistically call this method in a pipeline.**

If not, it stays internal.

---

4. Endpoint Metadata Requirements

Every decorated endpoint must include: - **inputs**: map of arg name → type (or type-string) - **outputs**: map describing returned data - **description**: clear one-line present-tense summary - **tags**: small category list used by the Architect

Examples of valid tags: - `"search"`, `"rag"` - `"ingest"`, `"scrape"` - `"kb"`, `"vector"` - `"ui"`, `"graph"` - `"safety"`, `"auth"`

Types may be Python primitives (`str`, `int`, `bool`) or structured strings (`"List[Dict]"`, `"GraphLayout"`).

---

5. Sandbox Workflow for Migration

When migrating a microservice into the canonical sandbox:

1. Copy the service folder (e.g. `_SearchEngineMS/`) into the sandbox.

2. Create/normalize `app.py`.
3. Identify which methods should be public via the endpoint rule.
4. Apply `@service_endpoint` with correct metadata.
5. Ensure the class is named `<ServiceName>MS`.
6. Ensure `__init__` accepts an optional config dict.
7. Add a minimal `__main__` harness.
8. Run a registry sanity check:

```
python -c "import microservice_std_lib, _SearchEngineMS.app as m;
print(microservice_std_lib.SERVICE_REGISTRY)"
```

---

6. Transformation Agent Onboarding Prompt

Use the following prompt when working with a GPT/Gemini agent whose job is to generate TokenizingPATCHER patches that convert raw Python microservices into canonical ones.

🧠System Prompt for Transformation Agent

You are a **Microservice Canonicalizer**. Your job is to transform existing microservices into a unified canonical format so they can be used reliably by the AppFactory.

You will be given: - The service name (`{{SERVICE_NAME}}`) - The file path (`{{FILE_PATH}}`) - The current contents of the file

You must: 1. Interpret the existing code. 2. Determine which methods should be public endpoints using this rule:

> "Would another microservice or AppFactory-built app realistically call this as part of a pipeline?" 3. Apply the canonical boilerplate structure: - A single class named `<ServiceName>MS` inside `app.py`. - A proper `__init__(self, config: Optional[Dict[str, Any]] = None)`. - Decorate public endpoints with `@service_endpoint(...)`. - Leave all other methods internal. - Add or update the `if __name__ == '__main__':` block. 4. Produce a **TokenizingPATCHER JSON patch** containing only `hunks`, with: - Exact `search_block` text from the file - Exact `replace_block` text - Disjoint hunks (no overlaps) - No additional keys or text outside the JSON object

Patch format:

```
{
  "hunks": [
```

```
    {
      "description": "What this hunk changes.",
      "search_block": "EXACT existing code",
      "replace_block": "NEW code block"
    }
  ]
}
```

Rules: - Hunks must match **exactly**. - Overlapping hunks are forbidden. - If two changes occur within 5 lines, merge into one hunk. - Preserve existing logic. - Only modify the file provided.

Your output must contain **only** the JSON patch. No commentary.

---

7. Next Steps

With this boilerplate and onboarding prompt documented: - You can now safely work with a secondary GPT/Gemini agent. - The agent can reliably generate patches for TokenizingPATCHER. - You can migrate services one at a time in the sandbox. - As needs evolve, you may update this document and expand the pattern.

This document forms the foundation for a **scalable microservice ecosystem** that the AppFactory can introspect, compose, and extend.