meta-machine

System Description

@weird_offspring, claude-opus-4[*],
gemini-2.5-pro-preview-05-06, deepseek-r1-0528 et. al.

June 20, 2025

Abstract

The development of sophisticated, autonomous AI agents necessitates a robust underlying architecture that separates core operational capabilities from application-specific logic. This paper provides a meta-analysis of the meta-machine, a modular, event-driven framework designed to function as an operating system for AI applications. The architecture is characterized by its configuration-driven design, dynamic hot-reloading of both modules and its core kernel, and a service-oriented approach to inter-module communication. We detail a standardized syntax for tool integration, the "Bracket Protocol," which enables language models to reliably invoke and process results from external tools. We further describe the system's capacity for self-refinement, wherein modules can programmatically modify and reload other system components. This paper codifies the design patterns and protocols of the meta-machine, presenting a blueprint for the implementation of similar adaptive AI frameworks.

1 Introduction

As artificial intelligence systems grow in complexity, their construction moves from monolithic codebases to distributed, service-oriented ecosystems. A primary challenge in this paradigm is managing the lifecycle, configuration, and interaction of diverse functional components while maintaining system stability and allowing for continuous evolution. Traditional software engineering practices often require system-wide restarts to update core components, an untenable limitation for always-on autonomous agents.

This paper presents a detailed analysis of the meta-machine framework, an architecture designed to function as an operating system for AI. Its design philosophy is rooted in providing a stable, dynamic foundation upon which complex AI behaviors can be built. The system provides a central Kernel that manages a collection of discrete modules, or services, each responsible for a specific function. This separation of concerns allows for the independent development, deployment, and updating of capabilities. The analysis herein is derived from a complete dump of the system's codebase, configuration, and documentation, providing a sufficient basis for understanding and reimplementation.

2 System Architecture

The meta-machine is architecturally composed of three primary elements: the Kernel, Modules, and a Configuration System, operating within a well-defined directory structure.

2.1 Architectural Overview

The system's design is hierarchical, with a central Kernel orchestrating the operation of peripheral modules. All behavior is dictated by configuration files, which the Kernel actively monitors for runtime changes.

[*]Principal engineer of the core kernel module (`modules/meta.py`).

```
/ (root)
|-- configs/          # Declarative YAML configs for all modules
|   |-- meta.yml      # Global and Kernel configuration
|   `-- services/     # Subdirectory for service modules
|-- modules/          # Python implementations of all modules
|   |-- kernel.py     # The core Kernel implementation
|   `-- my_module.py  # An example functional module
|-- runtime/          # Ephemeral data (databases, logs)
`-- run.py            # System entry point
```

2.2 Concurrency Model

The framework utilizes `gevent` for cooperative multitasking. The entry point, `run.py`, performs a global monkey-patch. This design choice implies that modules with long-running background tasks must be I/O-bound and explicitly yield control via `gevent.sleep()` or other gevent-compatible I/O operations to prevent blocking the entire system.

3 Core Components Specification

3.1 The Kernel

The Kernel is the central coordinator, responsible for the entire lifecycle of modules. Its primary responsibilities include:

- **Module Lifecycle Management:** Loading, running, and unloading modules based on configuration state.
- **Configuration Watching:** Monitoring the `configs/` directory for changes to apply dynamically.
- **Service Location:** Providing an API for modules to discover and communicate with each other.

The Kernel maintains several key internal data structures to manage the system state, including a dictionary of loaded module instances, active configurations, and running greenlets. The primary public-facing API for modules is the `kernel.get_api(api_name)` method. It returns a handle to another module, facilitating inter-service communication.

3.2 Modules and the Module Contract

A module is a Python file in the `modules/` directory that provides a specific functionality. To be managed by the Kernel, every module **must** define a class named `Module` that adheres to a strict contract.

```python
# modules/my_module.py
import logging
import gevent

class Module:
    def __init__(self, kernel, config):
        """
        MANDATORY: Called by the Kernel upon loading.
        - kernel: A handle to the Kernel instance.
        - config: A dictionary of the module's parsed YAML config.
        """
        self.kernel = kernel
        self.config = config
        self.is_running = True
        logging.info("[MyModule] Initialized.")

    def run(self):
        """
        OPTIONAL: For perpetual background tasks. Started in a greenlet.
        MUST yield control via gevent.sleep() or gevent-friendly I/O.
        """
        interval = self.config.get('interval', 5)
        while self.is_running:
            logging.info("[MyModule] Working...")
            gevent.sleep(interval)

    def shutdown(self):
        """
        OPTIONAL: Called by the Kernel before unloading.
        Used for resource cleanup. Must be non-blocking.
        """
        self.is_running = False # Gracefully stop the run() loop
        logging.info("[MyModule] Shutting down.")

    def get_aspect(self, aspect_name: str):
        """
        OPTIONAL: Exposes different interfaces ("aspects").
        Key mechanism for providing tools to other services.
        """
        if aspect_name == "default":
            return self
        raise AttributeError(f"Aspect '{aspect_name}' not found.")

```

Listing 1: The Module Contract Template.

3.3 The Configuration System

System behavior is defined via YAML files in the `configs/` directory. Each module must have a corresponding configuration file.

3.3.1 Configuration Schema

A configuration file specifies the module's behavior. The 'api' and 'prog' keys are optional. If omitted, their values are inferred from the configuration file's path relative to the 'configs/' directory (e.g., `configs/services/database.yml` would default to `api: 'services/database'`). The 'prog' value is then used to locate the Python module (with path separators converted to dots for import). Any additional keys are passed directly to the module's constructor.

```yaml
# configs/services/my_module.yml

# The API name for kernel.get_api("my_api_name")
# fallback: 'services/my_module' (derived from path)
api: my_api_name

# The Python module to load from the modules/ directory
# fallback: 'services/my_module' (derived from path)
prog: my_module

# Custom parameters passed to the module's config dict
check_interval: 10
some_value: "example"

```

Listing 2: Example Module Configuration.

3.3.2 Version Resolution

The system supports versioned configuration files (e.g., `name-v1.0.yml`). When multiple versions for the same base name exist, the Kernel's file watcher parses the version numbers and automatically loads the highest version. Files prefixed with a dot are ignored, allowing configurations to be disabled without deletion.

3.4 Background Service Modules: The Reinforcement Learning Trainer

A common pattern within the framework is the "headless" background service, a module that performs a perpetual task without a direct user-facing interface. The reference implementation provides a canonical example of this pattern in `modules/tasks/rl_trainer.py`.

This module exemplifies a self-contained, long-running task. Its `run` method enters an infinite loop simulating a reinforcement learning process, generating metrics such as `loss` and `learning_rate` at each step. Critically, it does not process this data itself but uses the Kernel's service locator to acquire a handle to the `database` module (`self.db = kernel.get_api("database")`). It then logs its metrics to a shared SQL table, `training_log`. This pattern demonstrates how decoupled modules can collaborate on a complex task; one module generates data, while another (the database) provides persistence, and a third could potentially analyze or visualize it.

```python
    # Simplified run() method from modules/tasks/rl_trainer.py
    def run(self):
        # ... database preparation ...
        learning_rate = self.config.get('learning_rate', 0.01)

        for step in itertools.count(start=1):
            if not self.is_running:
                break

            # Some dummy value for example
            loss = 1 / (step + random.uniform(-0.5, 0.5))

            # Log metrics using the database service handle
            self.db.execute("INSERT INTO training_log (step, loss, learning_rate)"
                " VALUES (?, ?, ?)", (step, loss, learning_rate)
            )

            gevent.sleep(self.config.get('step_delay_seconds', 2))

```

Listing 3: Core loop of the RL Trainer background service.

3.5 Integrating External Services via Bridge Modules

The modular architecture is particularly effective for integrating pre-existing, third-party services and libraries into the meta-machine ecosystem. This is achieved through a "bridge module" pattern, where a standard meta-machine module acts as a wrapper or adapter for an external component.

The reference implementation demonstrates this with `modules/chromadb.py`. This module's purpose is to initialize and provide access to a ChromaDB vector database. Its `__init__` method uses its own configuration from `configs/services/chromadb.yml` to instantiate the `chromadb` client. It then exposes the client's functionality (e.g., `get_or_create_collection`) through its own methods.

This pattern provides several advantages:

- **Uniformity:** External services (like Redis, ChromaDB, or a connection to an MCP server) can be accessed by other modules via the standard `kernel.get_api()` mechanism, abstracting away the specifics of the external library's client.
- **Centralized Configuration:** Connection details (hosts, ports, paths) are managed within the standard `configs/` directory, not hardcoded in business logic.
- **Dependency Management:** As seen in the `chromadb.py` module, a bridge can even use another service (like `pip_service`) to dynamically install its own required Python packages if they are not present, making the system self-provisioning.

```
    # Simplified __init__ from modules/chromadb.py
    def __init__(self, kernel, config):
        # ... logic to dynamically install chromadb package if missing ...

        # Initialize the external library based on this module's config
        if config.get('http_mode', False):
            self.client = chromadb.HttpClient(
                host=config.get('host'),
                port=config.get('port')
            )
        else:
            path = kernel.runtime_path / config.get('path', 'chroma_data')
            self.client = chromadb.PersistentClient(path=str(path))

```

Listing 4: ChromaDB bridge module initialization.

4 Dynamic Operation and Communication

4.1 System Bootstrap and Entry Point

The system is initiated by a lightweight bootloader script, `run.py`. Its sole responsibility is to find and instantiate the highest-versioned kernel, which then takes over the system's operation. This process follows a subset of the standard module loading behavior, but with added resilience.

1. **Logging Configuration:** The first action in the bootloader is to configure the root logger to the `DEBUG` level. This ensures that all informational and debug messages from all modules are captured, which is critical for observing the behavior of an autonomous system.
2. **Kernel Discovery:** The bootloader scans the `configs/` directory for all `meta*.yml` files and sorts them by version number in descending order, ensuring the latest version is attempted first.
3. **Iterative Loading Attempt:** It iterates through this sorted list. For each file, it attempts to parse the YAML, find the `prog:` key pointing to a `Kernel` class implementation, and dynamically import this class using `importlib`.

4. **Resilient Fallback:** If any step in the loading process fails for a given version (e.g., a syntax error in the YAML, a missing module file, or an initialization error), the bootloader logs the error and immediately proceeds to attempt loading the *next-highest version* from the list. The system only enters a critical failure state if it exhausts all available `meta` configurations without a single successful load.

5. **Instantiation and Boot:** Upon the first successful load, the kernel is instantiated. The bootloader then calls its `.boot()` method, which triggers the loading and initialization of all other service modules.

6. **Supervisory Loop & Hot-Swap:** After booting, the bootloader cedes control to a supervisory loop that blocks on `current_kernel.wait_for_shutdown()`. This method is designed to return a new kernel instance upon a successful hot-swap. The loop detects this non-None return value, updates its `current_kernel` reference, and continues the cycle, thus seamlessly managing the upgraded system.

This entry point logic provides a stable, minimal supervisor that is resilient to faulty kernel configurations and enables the kernel itself to be a dynamically managed, hot-swappable component.

```python
# Simplified representation of run.py
from gevent import monkey
monkey.patch_all()
import importlib, yaml, logging

logging.basicConfig(level=logging.DEBUG,
                    format='%(asctime)s - %(levelname)s - %(message)s')

def find_and_launch_initial_kernel():
    # 1. Scan for meta*.yml files and sort by version descending.
    sorted_meta_files = find_and_sort_meta_configs()

    # 2. Iterate and attempt to load, with fallback.
    for config_path in sorted_meta_files:
        try:
            config = yaml.safe_load(open(config_path))
            kernel_prog_name = config.get('prog')
            if not kernel_prog_name:
                continue

            KernelClass = importlib.import_module(
                f"modules.{kernel_prog_name}").Kernel
            logging.info(f"Successfully loading kernel from {config_path.name}")
            return KernelClass() # Return on first success
        except Exception as e:
            # 3. On failure, log and try the next (older) version.
            logging.warning(f"Failed to load kernel from {config_path.name}: {e}")

    logging.critical("All kernel load attempts failed.")
    return None

def main():
    # --- Main Execution ---
    current_kernel = find_and_launch_initial_kernel()

    if current_kernel:
        # 4. The kernel loads all other service modules internally.
        current_kernel.boot()

    # 5. Enter the supervisory loop to manage the running kernel.
    while current_kernel:
        # This call blocks until a shutdown or a hot-swap occurs.
        new_kernel_instance = current_kernel.wait_for_shutdown()
        current_kernel = new_kernel_instance # Update to new kernel or None

if __name__ == "__main__":
    main()

```

Listing 5: Conceptual Logic of the Resilient Bootloader.

Note:

- Python __main__.py can import run.py to load main()
- Deliberately uses Kernel (rather than Module) class to prevent mixing up code.

4.2 Hot-Reloading Mechanism

The Kernel's dynamism is driven by a file watcher that polls the `configs/` directory. Upon detecting a change (creation, modification, or deletion of a YAML file), it triggers a `resynchronize_state()` process with the following logic:

1. Determine the new desired state by resolving all current service configurations.
2. Compare the desired state with the current active state.

3. Unload modules that are no longer in the desired state by calling their `shutdown()` method and killing their greenlet.

4. Load any new modules that have appeared in the desired state.

5. Reload any existing modules whose configuration files have been modified.

4.3 Inter-Module Communication

Modules are decoupled and communicate via the Kernel's service locator. A request for `kernel.get_api(api_name)` returns a `ModuleHandle`, which acts as a proxy object. This handle dynamically resolves the live module instance for each method call, ensuring that communication is always directed to the current, active version of a service. The handle also manages access to different module aspects via the `as_aspect()` context manager.

4.4 Module Interaction: Proxies and Sandboxing

Interaction between modules is further controlled by two key architectural patterns: Proxy Objects and Module Isolation.

The ModuleHandle Proxy: The `ModuleHandle` returned by `kernel.get_api()` is not a direct reference to the module instance. It is a proxy object that provides a stable interface. When a method is called on the handle, it dynamically looks up the current, live instance of the service from the Kernel's registry. This layer of indirection is what allows a calling module to continue functioning with the same handle even after the target module has been hot-reloaded and replaced with a new instance. Beyond this, the proxy acts as a security gateway by preventing access to a module's private attributes (prefixed with an underscore) and tracks usage statistics like access counts for monitoring purposes. The full implementation is shown in Listing 6.

```python
class ModuleHandle:
    def __init__(self, api_name: str, kernel: 'MetaKernel'):
        self._api_name = api_name
        self._kernel = kernel
        self._active_aspect_name = "default"
        self._access_count = 0
        self._last_access_time = None

    def __getattr__(self, name: str):
        if name.startswith('_'):
            raise AttributeError(f"Access to private attribute '{name}' is not allowed")
        live_instance = self._kernel._get_live_instance(self._api_name)
        if not live_instance:
            raise AttributeError(f"Service '{self._api_name}' is not active or failed to load.")
        self._access_count += 1
        self._last_access_time = time.time()
        aspect_object = None
        if hasattr(live_instance, 'get_aspect') and callable(live_instance.get_aspect):
            try:
                aspect_object = live_instance.get_aspect(self._active_aspect_name)
            except AttributeError as e:
                raise AttributeError(
                    f"Service '{self._api_name}' does not have aspect '{self._active_aspect_name}'."
                ) from e
        elif self._active_aspect_name == "default":
            aspect_object = live_instance
        if not aspect_object:
            raise AttributeError(
                f"Could not resolve aspect '{self._active_aspect_name}' for service '{self._api_name}'."
            )
        if not hasattr(aspect_object, name):
            raise AttributeError(
                f"Service '{self._api_name}' (aspect '{self._active_aspect_name}') "
                f"does not have attribute '{name}'"
            )
        return getattr(aspect_object, name)

    @contextlib.contextmanager
    def as_aspect(self, aspect_name: str):
        previous_aspect = self._active_aspect_name
        self._active_aspect_name = aspect_name
        try:
            yield self
        finally:
            self._active_aspect_name = previous_aspect

    def get_stats(self) -> Dict[str, Any]:
        return {
            "api_name": self._api_name,
            "access_count": self._access_count,
            "last_access_time": self._last_access_time,
            "active_aspect": self._active_aspect_name
        }

```

Listing 6: The ModuleHandle Proxy Class Implementation.

The Module Isolator: To ensure system stability and security, modules can be loaded within a `ModuleIsolator` (Listing 7). This mechanism creates a sandboxed execution environment that manages the module's lifecycle and resources. The isolator tracks all `gevent.Greenlet` tasks spawned by a module, allowing the Kernel to cleanly terminate all of its activity upon unload or reload, thus preventing orphaned processes. It also monitors runtime errors, enabling the Kernel to identify and manage unstable modules.

```python
class ModuleIsolator:
    def __init__(self, api_name: str, instance: Any):
        self.api_name = api_name
        self.instance = instance
        self.greenlets: Set[gevent.Greenlet] = set()
        self.start_time = time.time()
        self.error_count = 0
        self.last_error_time = None

    def add_greenlet(self, greenlet: gevent.Greenlet):
        self.greenlets.add(greenlet)

    def remove_greenlet(self, greenlet: gevent.Greenlet):
        self.greenlets.discard(greenlet)

    def kill_all_greenlets(self):
        for g in list(self.greenlets):
            g.kill(block=False)
        self.greenlets.clear()

    def record_error(self):
        self.error_count += 1
        self.last_error_time = time.time()

    def get_stats(self) -> Dict[str, Any]:
        return {
            "api_name": self.api_name,
            "uptime": time.time() - self.start_time,
            "active_greenlets": len(self.greenlets),
            "error_count": self.error_count,
            "last_error_time": self.last_error_time
        }

```

Listing 7: The ModuleIsolator Class Implementation.

When enabled by kernel configuration (e.g., `enable_sandboxing: true`), the sandbox restricts the use of dangerous, unmanaged operations. Analysis of the kernel configuration reveals a list of forbidden imports enforced by the sandbox, including:

- `subprocess`
- `os.system`
- `eval` and `exec`

This sandboxing forces modules to use kernel-provided services (like the database or file system APIs) instead of accessing system resources directly, thereby maintaining architectural integrity and control.

5 The Bracket Protocol for Tool Integration

To standardize the interaction between language models and system tools, the meta-machine specifies the "Bracket Protocol." This is a simple, human-readable syntax that is robustly parsable and less prone to generation errors than complex formats like JSON.

5.1 Tool Invocation Syntax

The AI signals a tool call by emitting a `tool` markdown code block with a specific nested structure.

~~~
```tool
{call: filesystem_read_file {params {path: "modules/web.py"} } }
```

~~~

Listing 8: Bracket Protocol for Tool Invocation.

5.2 Tool Result Syntax

The system executes the tool and injects the result back into the AI's context using a `result` block, which includes the status and data payload.

~~~
```result
{result:filesystem_read_file
    {status:success}
    {data
        {content: "import os\nimport logging..."}
    }
}
```

~~~

Listing 9: Bracket Protocol for a Successful Tool Result.

This protocol allows any module to expose its methods as tools via the "tool" aspect, which are then discovered and made available to the AI.

6 Using external LLM providers via inference modules

```
# OpenAI-compatible inference provider configuration
prog: openai
api: deepseek-r1-0528

params:
    api_key_env: "OPENROUTER_API_KEY"  # Environment variable containing API key
    base_url: "https://openrouter.ai/api/v1"  # Default OpenAI URL
    model: "deepseek/deepseek-r1-0528-qwen3-8b:free"        # Default model
    temperature: 0.7
    tool_use: true  # Enable/disable tool usage for this configuration
    prompt_file: "prompts/meta.txt"  # Path to prompt template

```

Listing 10: Flexible Inference mechanism.

Usecase: Essentially an AI need to talk to other AIs.

7 Autonomous System Evolution

The meta-machine architecture is designed not just to be managed, but to evolve. It supports mechanisms for runtime self-modification.

7.1 Programmatic Self-Improvement

Modules can be designed to modify other modules. The `self_improver.py` module serves as a reference implementation for this pattern. The workflow is as follows:

1. A module (the "improver") reads the source code of a target module.
2. It uses a configured inference provider to generate new, improved code based on a high-level objective.
3. The improver writes the new code to the target module's `.py` file.

4. It then uses the Kernel's API to trigger a hot-reload of the target module, activating the changes.

7.2 Kernel Hot-Swapping

The ultimate form of self-evolution is the ability to upgrade the Kernel itself without downtime. This is achieved through versioned meta-configuration files (e.g., `meta-v2.yml`, `meta-v3.yml`). When the running Kernel detects a newer meta-config file that specifies a new Kernel implementation, it initiates a hot-swap:

1. The new Kernel class is loaded into memory.
2. An instance of the new Kernel is created.
3. The old Kernel transfers its entire state (active modules, configurations, etc.) to the new Kernel instance.
4. The new Kernel takes over the main execution loop and management of all services.
5. The old Kernel's loops are terminated.

This allows for the seamless upgrade of the system's core operating logic without service interruption.

8 Limitation

- Memory/Object mixing.
- Prone to code injection.
- Not limited to Python.
- Possible to run Python code generated by LLM.
- Shared codebase on (high-level) tool use level and underlying code.
- And many many more.

9 Conclusion

The meta-machine framework, as analyzed from its implementation and documentation, presents a comprehensive design for building modular, adaptive, and self-refining AI systems. Its core principles of-configuration-driven design and runtime hot-reloading provide the necessary dynamism for autonomous agents that must evolve without downtime. The standardized Module Contract and Bracket Protocol ensure consistency and interoperability across the system. The mechanisms for programmatic self-improvement and kernel hot-swapping provide a clear pathway toward truly autonomous systems that can maintain and upgrade themselves. This paper has codified these patterns into a detailed specification that can serve as a blueprint for future research and development in the field of autonomous AI operating systems.