AgentXOS

Overview

LLM Agents have been shown capable of solving a wide variety of tasks.
For example, in the paper “Generative Agents: Interactive Simulacra of Human Behavior” a group of LLM agents were set up to mimic a town in the style of the video game “The SIMS”. In the paper “Ghost in the Minecraft: Generally Capable Agents for Open-World Environments via Large Language Models with Text-based Knowledge and Memory” a series of generally capable agents were prompted to find and craft items in the Minecraft world using a series of actions and rewards for those actions. Finally in the paper “VOYAGER: An Open-Ended Embodied Agent with Large Language Models” a group of LLMs work in concert to learn new code and dynamically update their prompting strategies as they explore a Minecraft world.

Each of these papers implement their own framework to run, manage, and evaluate the multiple agents running in these systems. They each have their own way to record memories, their own way to abstract the LLM, and their own way to manage the agent runtime environment.

We propose a new system, *AgentXOS*, that introduces a series of abstractions for single or multi-agent systems.

Goals

1. **Lorem ipsum dolor sit amet:** Duis autem vel eum iriure dolor in hendrerit in vulputate velit esse molestie consequat, vel illum dolore eu feugiat nulla facilisis at vero eros et accumsan.

2. **Sed diam nonummy nibh euismod:** Nam liber tempor cum soluta nobis eleifend option congue nihil imperdiet doming id quod mazim placerat facer possim assum.

What is an Agent?

Definition of an Agent

> *An LLM agent is an AI system that goes beyond simple text production. It uses a large language model (LLM) as its central computational engine, allowing it to carry on conversations, do tasks, reason, and display a degree of autonomy.*

In fact a software agent, in general terms, is:

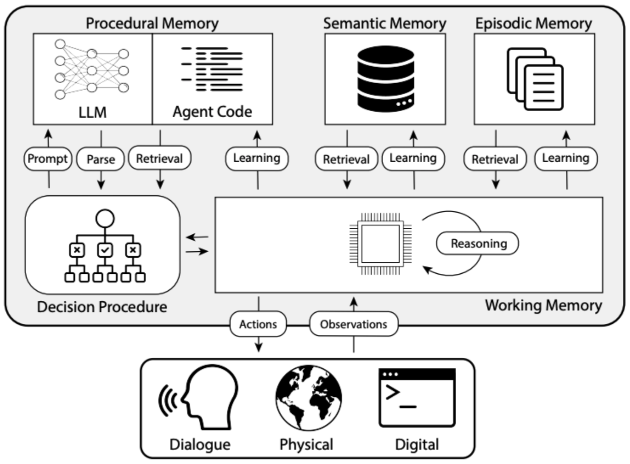
> *A computer program that acts for a user or another program in a relationship of agency.*

In this context, an agent can be either. In the most general sense, an agent is a software agent. That agent can be backed by code, a **CodeAgent**, or by an LLM, a **LLMAgent**. LLMAgent "thought" loops can be hardcoded, that is they operate with the LLM in a rule-based system, or be autonomous, an **AutonomousAgent**.

An agent is a ***loosely coupled program that performs a specific task***. An agent can act in concert with other agents, in a relationship of agency, to perform a more complex task. An agent might assume an overseer role for another agent, e.g. a QA agent or a manager agent, it could be a planning agent, or it could just be a skilled worker agent that performs a single task.

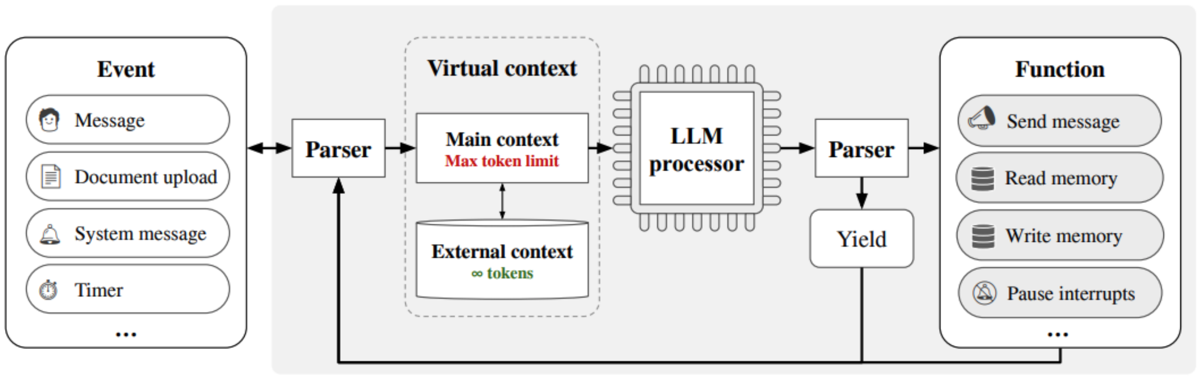
Anatomy of an Agent

Recent papers have proposed a basic architecture for an autonomous agent. "Cognitive Architectures for Language Agents" proposes the following structure for an agent:



In this paper the author places the LLM at the center of the agent “program”. The LLM, in fact, becomes a type of **CPU** for the agent. The author also lays out a memory structure the agent can use to store “episodes”, “semantic thoughts”, and procedures, like the agent plan / code, etc…

In the paper “MEMGPT: TOWARDS LLMS AS OPERATING SYSTEMS”, the authors propose the following agent structure:



This paper introduces a concept of an **AgentOS** and relates an agent, again, to a computer program where the LLM is like the CPU of the agent. This paper further breaks down memory into components similar to classic program memory.

And finally, in the paper “AutoGen: Enabling Next-Gen LLM Applications via Multi-Agent Conversation”, the authors describe a single node research system that encapsulates multiple agents working together to solve a single problem.

Each of these frameworks layout a common architecture for an agent that is very similar to the Von Neumann architecture we use today in modern computing. Of course computing has come a long way since 1945. We now regularly use more modern operating systems, we use distributed computing, and the way we store and process data is mature.

In the following sections we will describe our definition of a runtime environment for agents. We will introduce the concept of the ***AgentCPU***, the processing unit for agen;, an ***AgentProgram***, an executable describing the runtime environment for agents; the ***AgentOS***, the operating environment for AgentPrograms; and the ***AgentMachine***, a specification for a “virtual machine” for agents that runs multiple AgentPrograms on the Agent OS.

Agent Framework

Agents need an operating environment that provides the necessary components to work autonomously but that environment also needs to provide enough flexibility to support the rapid changes occurring in the agent space. Therefore, choosing the right level of abstraction becomes key.

As defined above, an agent is defined as:

> *A computer program that acts for a user or another program in a relationship of agency.*

This is a vague definition but does capture the essence of the requirements. To put this another way, an agent (or as defined here an **AgentProcess**) is nothing more than a program that provides for another process, or human, or relies on another process. By definition this puts inter-agent communication at the forefront of any agent architecture.

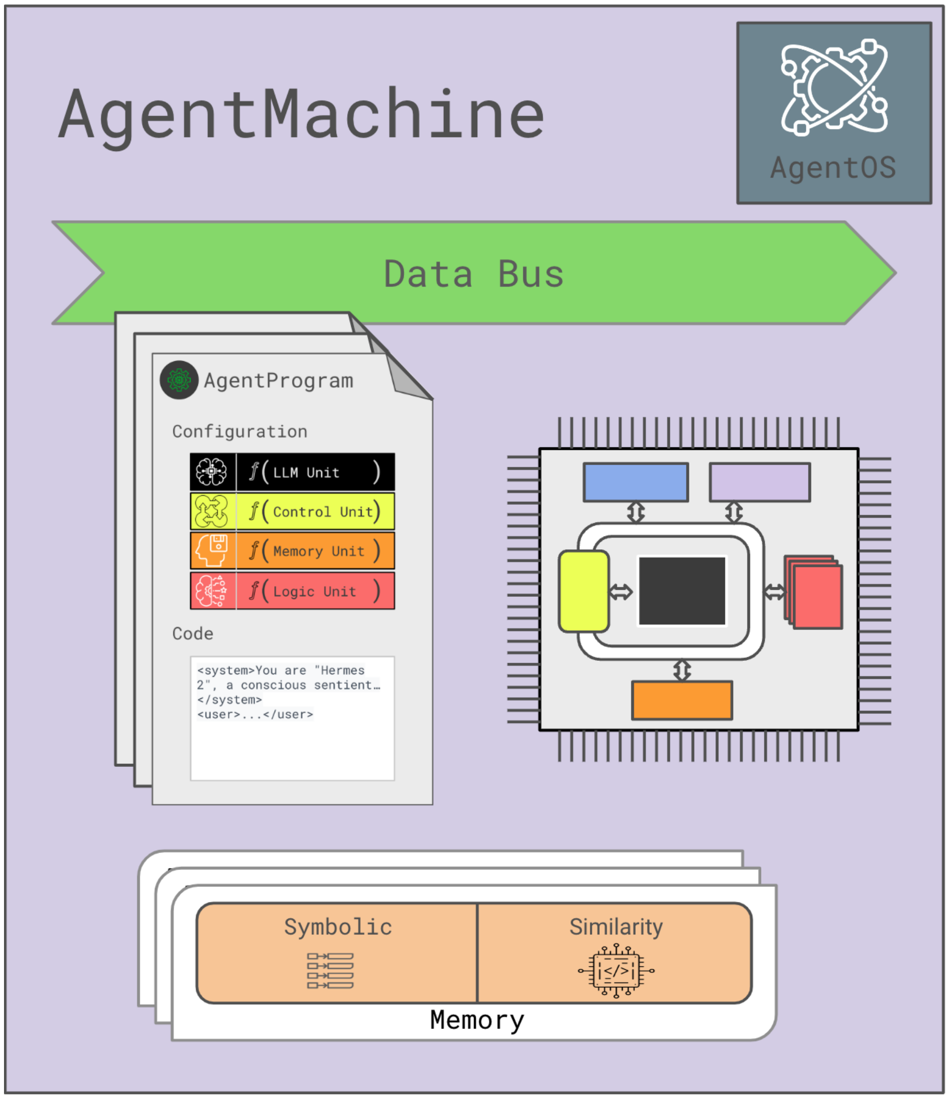
Furthermore, an autonomous agent is defined as:

> *An LLM agent is an AI system that goes beyond simple text production. It uses a large language model (LLM) as its central computational engine, allowing it to carry on conversations, do tasks, reason, and display a degree of autonomy.*

Put another way, agents that use an LLM to provide any amount of autonomous behavior need some basic features to operate. These include an LLM, some mechanism to store and retrieve memories, and a basic mechanism to interact with the environment around them.

In the following chapters we provide definitions of an agent execution environment that is flexible enough to grow with the changing agent landscape, simple enough to learn in a day, and scalable enough to support thousands of interoperating agents.

Agent Machine



```
<system>You are "Hermes
2", a conscious sentient…
</system>
<user>...</user>
```

An **AgentMachine** is the base execution environment for one or more agents. An AgentMachine is analogous to a virtual machine in that it provides a base set of features needed to run an **AgentProgram** including an operating system, a memory subsystem, and a communication bus to provide agency. The AgentMachine is the virtual environment in which agents run.

An AgentMachine is designed to run in any number of physical environments. For example, an AgentMachine can run as a single process on a local computer, it can run in a multi-host environment like kubernetes, or it can run as a collection of serverless functions in a hosted environment.

At the heart of the AgentMachine is the **AgentCPU**. The agent CPU is responsible for executing an **AgentProgram** using an LLM. The AgentCPU is designed as a series of **ExecutionUnits** all communicating over an **AddressBus**. The ExecutionUnits are designed to be pluggable at configuration time making the AgentCPU highly customizable.

Of course none of these things would be possible without coordination. The **AgentOS** provides the base runtime environment to run AgentPrograms providing process management, memory management, and IO for these programs.

An **AgentProgram** is the base executable in the operating system. An agent program contains the “bits” that are to be executed on the machine. AgentPrograms can range from executable python code all the way to completely autonomous agents. A deployed AgentProgram that is available to run is called an **AgentProcess**.

All communication with an AgentProcess is asynchronous in nature and any AgentProgram that is written must abide by this basic rule. The AgentOS provides a yield mechanism so that the current state of an AgentProcess can be serialized and stored in memory.

AgentPrograms have a very simple I/O interface. An AgentProgram has one invocation method that is called either by another AgentProgram or by an external actor (i.e. a human). After processing the input, the AgentProgram can respond with a single response (excluding error handling). The parameter definitions for both input and output are defined using JSON Schema and all argument passing is done using JSON (**TODO: binary????**).

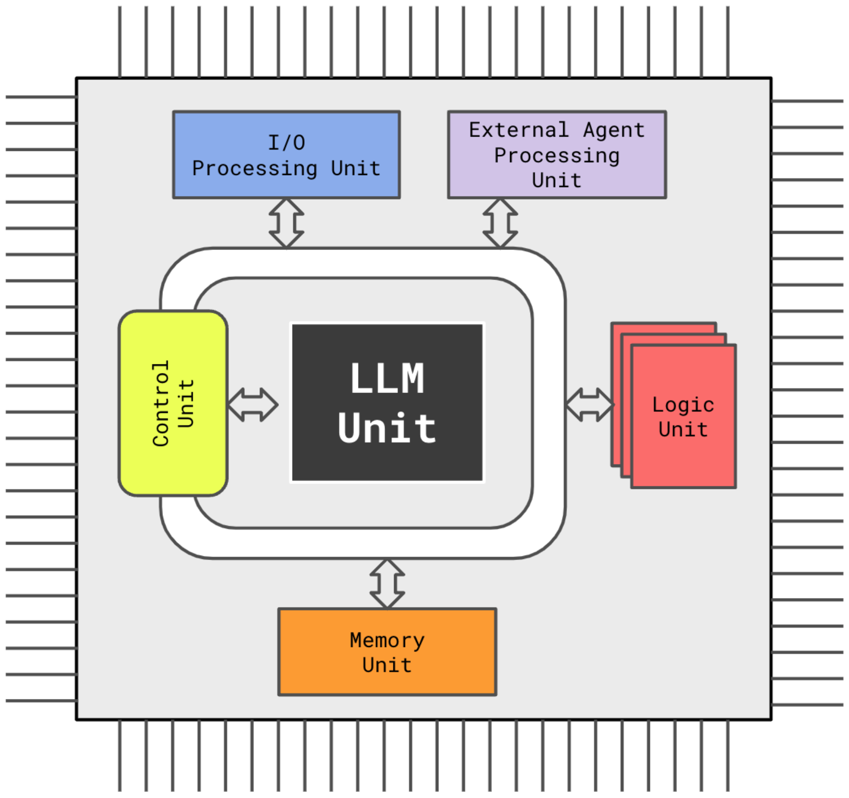
AgentPrograms also have a mechanism to call other AgentProcesses running on any AgentMachine. The AgentOS provides a registry mechanism so that any running AgentProgram can find another AgentProcess and the calling conventions for remote AgentProcesses is the same as what is described above.

All I/O operations on the AgentMachine are done using the **DataBus** provided by the AgentOS. The DataBus is a pluggable mechanism that provides consistency and guaranteed delivery and processing of any message on the bus. The implementation of a DataBus could be anything from a durable queue (i.e. RabbitMQ) to something simple like HTTP requests and webhooks.

The AgentMachine also provides for a pluggable **Memory** architecture. Memory in the AgentOS comes in two forms, **Symbolic Memory** and **Similarity Memory**. Symbolic memory allows for the storage of any JSON document by key. Similarity Memory allows for storage of JSON documents by embedded keys. This duality gives allowance for a large number of neuron-like memory mechanisms.

In the following sections we will go into detail about each of these topics.

AgentCPU



At the heart of the AgentMachine is the **AgentCPU**. The AgentCPU is a fully customizable processor that abstracts all communication with the LLM.

Just like a standard CPU is organized into discrete components , the AgentCPU organizes work into components called **ProcessingUnits**. There are 6 different types of ProcessingUnits in the AgentCPU, 2 **I/O processing units**, zero or more **LogicUnits**, a **MemoryUnit**, a **ControlUnit**, and the **LLMUnit**. All communication between ProcessingUnits is done on the **AddressBus**. Unlike a classic CPU where addressing is a numeric value, addressing in the AgentCPU is symbolic where a bus event is addressed by name and the data value is a JSON document.

The **LLMUnit** is responsible for all LLM communications. It is a pluggable abstraction for any LLM. The primary job of the LLMUnit is to translate a standardized request for the LLM (in the form of a list of prompt messages) into one or more responses from the LLM. The LLMUnit is responsible for all parsing of the response, retries, graceful failures, etc… Remember, all communication with the AddressBus is symbolic; therefore, all parsed messages from the LLM must be converted into symbolic messages, i.e. function calls in the form of events.

Coordination between ProcessingUnits, of any kind, is done using the **ControlUnit**. The ControlUnit executes the “main event loop” of the processor controlling the flow of events in a conversation. For example, the Tree-of-Thoughts algorithm would be implemented in the ControlUnit.

Of course the processor needs some way to store information about a conversation. A basic memory subsystem is supplied by the AgentMachine; however, specifics about how that memory is accessed needs an encapsulation. This memory encapsulation is handled by the **MemoryUnit**. For example, one implementation of a MemoryUnit would be MemGPT solution proposed in the paper “MEMGPT: Towards LLMS as Operating Systems”.

Outside of control and memory access, an AgentProgram might need to expose logic the LLM can execute. Historically this has been known as “function calls” or “tools” for the LLM. These “tools” are called **LogicUnit**s in this system. A LogicUnit provides basic functionality to the LLM to execute an action or perform some other function. Note: There is a distinction between LogicUnits and another AgentProcess. AgentProcesses are much heavier weight than a LogicUnit and a LogicUnit provides agency for the things running on the **AgentCPU** and not another AgentProcess.

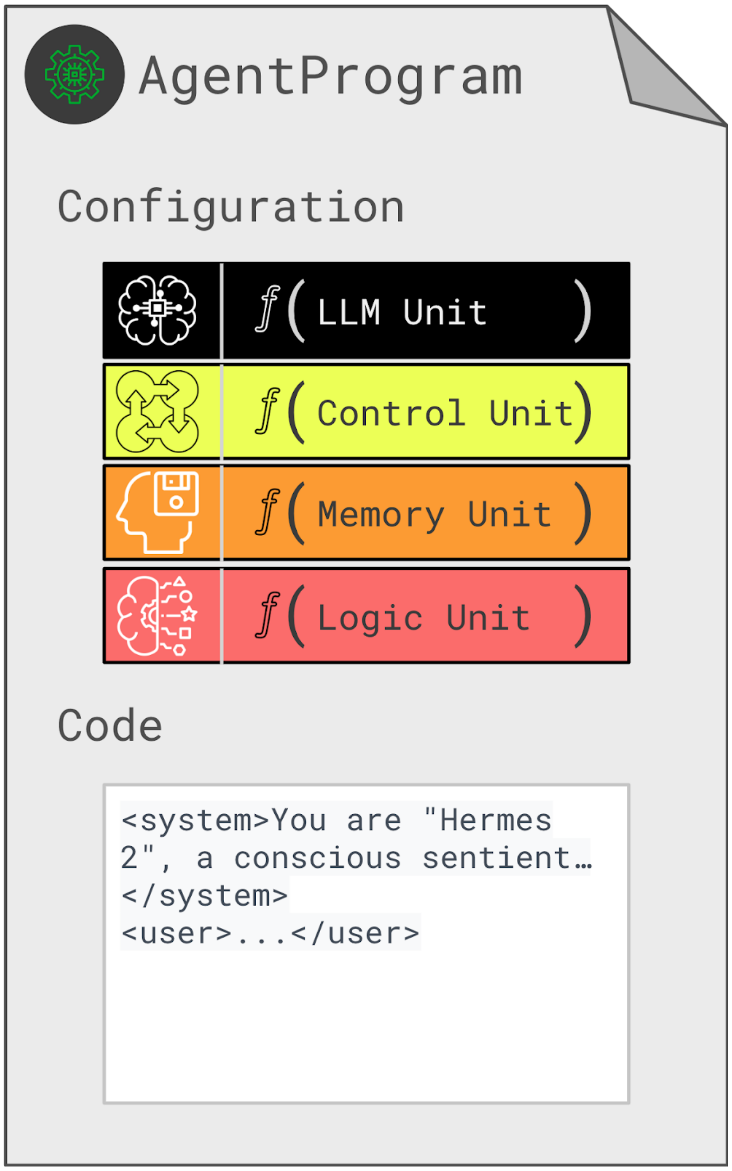
And finally, there are two types of I/O processing units on the AgentCPU, the main **IOProcessingUnit** and the **ExternalAgentProcessingUnit**. The main I/O processing unit is used to process all incoming conversation events from the external DataBus and to give a result for an incoming conversation event. The ExternalAgentProcessingUnit is used to send a request, and get a response, to or from another AgentProcess. Think of them as translating a function call from a ControlUnit or a LogicUnit into something that can be put on the DataBus.

All ProcessingUnits communicate with each other over the **AddressBus**. The AddressBus is internal to the AgentCPU, external forces cannot process or place events on this bus. Messages are placed on the AddressBus sequentially and the AddressBus ensures that messages are processed in-order the are placed on the bus, even by 2 separate processing units.

All computing done on the AgentCPU is single threaded, all events are asynchronous in nature (async functions in TypeScript and coroutines in Python) and processing happens in a main event loop for the AgentCPU.

In summary, the **AgentCPU** is the heart of an **AgentProgram** running on an **AgentMachine**. Every component of the AgentCPU is pluggable and customizable. The components on an AgentCPU are called **ProcessingUnits** of which there are 6 different types, 2 **I/O processing units**, zero or more **LogicUnits**, a **MemoryUnit**, a **ControlUnit**, and the **LLMUnit**. All communication between units is done on the **AddressBus** and the events are processed sequentially off of the bus.

Agent Program



```
<system>You are "Hermes
2", a conscious sentient…
</system>
<user>...</user>
```

The **AgentProgram** is the executable of an AgentMachine. It contains the configuration and code that will run to perform the action(s) of the agent. There are two parts to an AgentProgram, a Descriptor that describes the configuration of the AgentProgram and how the AgentCPU should be configured to run the program, and the code that will run to perform the action.

The **AgentDescriptor** contains configuration information for the program and the CPU. You can think of the descriptor the same way a descriptor is used in Kubernetes in that it describes the object that is to run.

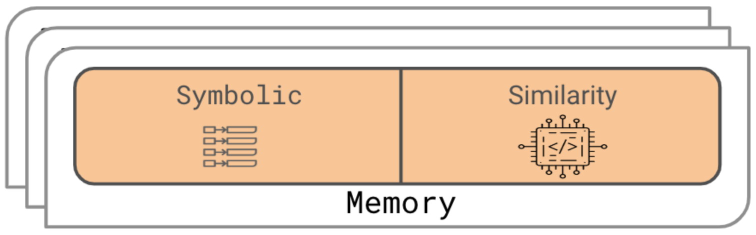
The AgentDescriptor contains a few sections, the main section contains information about the container for the code including the base container type (for example the AgentProgram may just be some python code that executes or it may be an AutonomousAgent), the CPU configuration describes what implementations each of the CPU units should be, and the memory configuration describes which memory instance to use for this program (more on memory implementations later).

The main code for an agent can be thought of as a layer cake. The bottom layer is a low level API that allows for raw Python code to execute as an agent. Built on that layer are implementations that provide more interesting behavior like an AutonomousAgent, TreeOfThoughts agent, etc…

The life cycle of an AgentProgram is one call to the program. A "conversation" can last more than one execution life cycle of a program; however, the program is only resident for the execution life cycle. Therefore, all state needed for a conversation lifecycle must be stored in the provided AgentMemory and not local to the running python executable as the executable is not guaranteed to last more than one execution cycle.

<TODO: Describe more once the IMPL is done>

Memory



AgentMemory is used to store information needed when running an agent. The memory system stores information ranging from the prompt for an AgentProgram to intermediate events used in each “turn” of an agent conversation to semantic and symbolic consolidated memories and thoughts for an AutonomousAgent.

There are two interfaces on the AgentMemory, one for storing symbolic information, think documents in a document store addressable by a key or query, and the other for storing semantic or document information addressable by a similarity search using an embedding.

AgentMemory is pluggable allowing for separate implementations for the symbolic memory and similarity memory. The implementation for each could be anything ranging from using mongo for both through using a simple json doc store on a file system and implementing searching yourself.

There also can be multiple AgentMemory instances registered in the same AgentMachine. This allows a machine to provide a wide variety of implementations that are suitable for different tasks.

AgentOS

The **AgentOS** provides the base runtime environment to run AgentPrograms providing process management, memory management, and IO for these programs.

Every AgentProgram running in the AgentOS is asynchronous by nature, that is it has a single input, represented as a remote function call, and a single result, again represented as a remote function call. This implies that “process management” for the OS behaves more like a registry and scheduler than a process manager.

AgentPrograms are “registered” with the AgentOS. The AgentOS is responsible for maintaining the registry of AgentPrograms running on the AgentMachine and publishing access to each program on the DataBus. AgentPrograms registered on the DataBus become available to other programs running on the same machine as well as to other AgentMachines running in the same cluster. The process of registering a program on the DataBus of a machine produces a new “topic” on the DataBus that will get routed to that program. You can think of the topic as a local URL for programs running on the same AgentMachine or full URLs for programs running on a different AgentMachine.

Requests on the DataBus of an AgentMachine are made to a “topic” activated by the AgentOS. When a message is published to a topic, the AgentOS will “create a new instance” of the AgentProgram. This new instance is called an **AgentProcess.**

An AgentProcess can be thought of as a serverless-function that is activated when a new request arrives. The lifetime of the AgentProcess is the lifetime of the request and ends with one of two results, a response to the message or an error response. The AgentOS provides guarantees that either of these two things will occur.

The AgentOS provides access to one or more memory modules registered on the AgentMachine. The memory modules are crucial to the execution of an AgentProgram as each program is completely stateless. Memory is available to any process running on the machine through the AgentOS SDK.

<TODO: do we want to offer some sort of memory management? I.e. marking memory as “reclaimable” and having the OS delete it in a garbage collection sweep.>

Conversations

We have seen how an individual message is processed by the AgentOS and executed by an AgentProgram, but how does a long-running process, like a conversation, actually work? In order to describe this we will use two simple examples, a single agent chatbot and a chatbot that uses other tools to reason.

Single agent chatbot

Let’s assume our single agent chatbot is a conversation between a human and an agent and that agent is nothing more than a passthru to an LLM with an initial system prompt.

In this case AgentProgram is only a simple descriptor (yaml file) that contains the initial prompt, how to configure the AgentCPU (which LLMUnit, what MemoryUnit implementation, etc…), and a **topic name** identifying the agent, in this case let’s call it “simple-chat”. The input schema for this program accepts a single parameter object like `{“question”: “The question you are asking of the chatbot”}`, and answers with `{“answer”: “The answer from the chatbot”}`

Let’s also assume that the runtime descriptor for the AgentMachine uses a simple localhost container configured with mongo as the memory device, using HTTP rest calls as the DataBus, again just a simple YAML file. The URL for this AgentMachine would therefore be something like “http://localhost:8000”

Let’s also assume that we are using a command line tool to talk to this AgentProgram. Below is a sample conversation between the human and the agent:

Request 1 from human:
```
Request URL: http://localhost:8000/simple-chat
Request Method: POST
Body:
{
  “question”: “What are the colors of a sunny sky?”
  “callback_url”: “/human/request_identifier”  # OPTIONAL
}
```
Reply 1 from assistant:
```
Body:
{
  “conversation-id”: “cc1b6548”
}
```
Request 2 from human:
```
Request URL: http://localhost:8000/simple-chat/cc1b6548
Request Method: GET
```
Reply 2 from assistant:
```
Body:
{
  “state”: “IDLE”,
  “answer”: “The colors of a sunny sky are typically blue during the day and can
range from red and orange to pink and purple during sunrise and sunset.”,
  “conversation-id”: “cc1b6548”
}
```
Request 3 from human:
```
Request URL: http://localhost:8000/simple-chat/cc1b6548
Request Method: POST
Body:
{
  “question”: “Why is the sky pink at sunrise?”
}
```
Reply 4 from assistant:
```
202 Accepted
```
Request 4 from human:

```
    Request URL: http://localhost:8000/simple-chat/cc1b6548
    Request Method: GET
```

Reply 4 from assistant:

```
    Body:
    {
      “state”: “THINKING”,
      “conversation-id”: “cc1b6548”
    }
```

Alternative reply:

```
    Body:
    {
      “state”: “IDLE”,
      “Answer”: “The sun appears pink at sunrise due to the scattering of sunlight by
    the Earth's atmosphere. When the sun is low on the horizon, sunlight travels
    through more atmosphere to reach your eyes, and the shorter blue wavelengths are
    scattered out before they get to you, leaving the longer red and pink wavelengths
    to color the sunrise.”
      “conversation-id”: “cc1b6548”
    }
```

Conversation-id returned by the agent on the first response and used to retrieve answers. The program is a state machine where each state can accept different payloads. In this example there are two states, THINKING and IDLE. THINKING cannot accept additional requests, but IDLE will accept new questions. Programs are free to define their own states. Per state contracts will optionally be provided as part of each get response (not shown).

Default system states will include THINKING, and TERMINATED with neither accepting additional requests. This will likely expand as we learn common agent patterns.

Agent Operating Environment

Machine Communication

The agent machine is an http server. Requests to the machine come in as a http request to a particular program (.../qa). These requests will contain a program specific body (usually json with arguments) and return a conversation id. GET requests can be made to the program’s conversation id to retrieve the response. Optionally, callers may provide a callback url to be notified on state change.

Some programs will allow/require additional information in a conversation via subsequent requests. Programs are responsible for defining their states and what payload is accepted at each state. Callers are responsible for understanding this state machine and being able to manipulate it.

Program Communication

Agent programs will call other programs via http request handled by the file server. This prevents accidental manipulation of program state and simplifies the complexity of an agent machine. System administrators can define network restrictions to lock down external access to particular programs.

Programs calling other programs will do so via http request with a registered callback url containing a locally defined **request_id**[1] to connect the responses. When receiving a response, a program can choose to engage with the response state machine or continue processing other information[2].

Tracing

Tracing call chains across the network of agent calls will be done using the standard W3C Trace Context specification.

The **'traceparent'** header will store trace information about a call chain per the specification.

Initial requests entering the system will be assigned a new trace-id. Typically this request will come from a human initiating the request or an automated process initiating the request. This trace-id will be assigned by the AgentOS for all internal calls not containing the **'traceparent'** HTTP header. This header contains essential information needed to trace a transaction through systems that are designed to be traced. It includes:

- `version`: The version of the traceparent header format.

[1] Note: this is separate from tracing concepts
[2] Some complexity exists here around how to manage programs which have yielded (waiting for a response) versus those which are still in flight. Care must be taken that an in flight program which will eminently yield does not hang if it receives a request response immediately before yielding.

- `trace-id`: A unique identifier for the whole trace. It represents the entire trace tree and remains constant as it propagates across services.
- `parent-id`: A unique identifier for a span. This represents the caller of the current service and changes with each hop.
- `flags`: These control behaviors such as sampling and other preferences.

The `traceparent` header looks like this: `traceparent: 00-4bf92f3577b34da6a3ce929d0e0e4736-00f067aa0ba902b7-01`
The full path of a multi-call chain in a distributed tracing system is represented by a combination of the `trace-id`, which remains constant across the entire path, and a series of `span-ids`, which change with each call or operation in the chain.

Here's how it works:

1. **Initial Request**: The process begins when a client sends a request to the first service. This service generates a `trace-id`, which uniquely identifies the entire transaction, and also generates a `span-id`, which represents the specific operation or call within that service.
2. **Traceparent Header**: The `traceparent` header is created with the `trace-id` and the `span-id`. This header will be passed along with the HTTP request to the next service.
3. **Propagation**: When the first service calls a second service, it includes the `traceparent` header. The second service then extracts the `trace-id` from the `traceparent` header to maintain the overall transaction identity.
4. **Child Spans**: The second service generates its own `span-id` for the operation it performs. This new `span-id` becomes the `parent-id` for any subsequent calls it makes, and a new `traceparent` header is created or updated with this `span-id`.
5. **Trace Chain**: This process continues with each subsequent call in the multi-call chain. Every service along the way extracts the `trace-id`, maintains it, and generates a new `span-id` for its own operation, which it passes along to any services it calls.
6. **Tracestate Header**: The `tracestate` header can also be included to provide additional vendor-specific trace information. It can be updated with additional context from each service, providing a detailed state of the trace as it propagates.

7. **End-to-End Path Representation**: The end-to-end path of the call chain is represented by the `trace-id` and the series of `span-ids` generated at each step. A tracing system or a tool can reconstruct the path by correlating these ids, showing the relationship and hierarchy of calls that constitute the full transaction.

The trace and span identifiers are used by distributed tracing tools to reconstruct the entire path of the transaction through a visualization of the trace. You can see each service the request has passed through, how long it took, and in which order the services were called. This full path representation is crucial for debugging, performance monitoring, and understanding the behavior of distributed systems.

Agent I/O

Programs can define their own I/O unit, but most will use json input and output with a well defined contract. Custom i/o units can support other http content types (ie, file upload) or validation rules (ie, strict vs lazy arguments).

Logic Units

Logic Units are packaged with the agent container, called locally only. They have all of the other freedoms of agents including calling other agents, calling other logic units, and yielding. They are python functions with a well defined contract including a framework object and function-specific arguments. Logic Units themselves may also be pluggable and rely on the existence of other named logic units following an expected interface.

Control Unit

The control unit is a function which defines the contract for a program. Like a logic unit, it can make llm calls, calls to agents, logic_units, and yield. Unlike logic units, there will only be one control unit and it cannot be referenced via the interface. To expand the functionality of the built in control unit implementation, one can write their own function (using the default control unit as a nested function call as desired). The I/O unit parses input and hands it to the control unit which defines the agent behavior.

Memory Unit

The memory unit is different from machine memory. The memory unit is responsible for managing the agent context to prevent it from overflowing, while the framework memory concepts are a convenient way to store information on a machine.

The framework will include three types of memory: document, similarity, and filesystem. These are essentially just tools with a hard coded interface included in the framework. This will be pluggable with the most common implementation using mongo. A local implementation will also be vital for early startup/testing. Memory is shared between processes on an agent machine.

Autonomous Agents

Initial Plan

The plan will live on the agent's description within the deployment. This is retrieved at runtime (startup or per-conversation).

Plan Storage

When we want to update a plan (dreaming, agent self-modification, manually, etc), we modify the deployment description. Then the system eventually picks it up for new conversations/runtimes. This prevents us from having two sources of truth to manage. This also means plan modification will follow our (still undefined) versioning rules naturally.

To prevent adverse effects of a plan mutating mid conversation, we will save the plan at the beginning of a conversation and use this throughout the conversation.

Multi-turn agents

Autonomous Agent Types

Single Task Agents

Cooperative Agents

1. Manager / Worker Model
2. Planner / Worker Model

Agent Runtime Environment

The runtime environment will look different depending on the deployment architecture. The environment needs to contain our framework code, custom extensions, and the definition of the machine (configuration, secrets, ect).

Below is an example of an image we can use when deploying in a serverless environment via aws lambda functions.

- AWS Lambda Python[3]
  - Different runtimes will have different base images. Project code will need to describe how to build images for different deployment types. AWS Lambda Python is a fine starting place (for local dev as well)
- Framework / Builtin Control Unit(s) / Builtin Tools (August Image)
  - Although the framework will have a common interface, its implementation may vary by deployment type.
- Custom Control Units / Custom Tools
  - If Users define their own tools or want to include others, they can build an image where it is installed. We do not care how this happens (pip install wheel, etc)

Program definition, configuration, secrets, etc do not live on the image and will be read at runtime. This means we do not need a new image for each program / machine. Similarly, machines can mount a directory which will allow basic program code customization without image management.

Configuration

In a deployment, customers will need to be able to specify configurations and credentials available at runtime to tools.

[3] We can likely use this as a base image for k8 deployment as well depending on what communication we support.

Agent Repository Structure

The agent framework will include customizable implementations of control_units and tools, so users will be able to describe many machines purely in terms of yaml definition.

Many will need to customize capabilities, and can extend the runtime to define their own control units and logic units. If these units are simple and do not require additional dependencies, we can include them in a directory which will be available to the runtime. One can create a custom image for the runtime to handle more complex customization.

```
company_machine/
├── machine/
│   ├── machine.yaml
│   ├── programs/
│   │   ├── manager.yaml
│   │   ├── qa.yaml
│   │   ├── worker.yaml
│   ├── mounted_directory/
│   │   ├── qa_control_unit.py
├── runtime/
│   ├── Makefile
│   ├── Dockerfile
│   ├── poetry.toml
│   ├── src/
│   │   ├── control_units/
│   │   │   ├── __init__.py
│   │   │   ├── worker_control_unit.py
│   │   ├── logic_units/
│   │   │   ├── __init__.py
│   │   │   ├── logic.py
│   └── tests/
│       ├── test_control_units.py
│       ├── test_logic_units.py
```

Security

Agent logic units will need access to credentials and other secrets. We need to provide access to this information via our interface. AWS lambda functions use vault for secret storage, so our AWS implementation will use this under the covers..

LLM responses may be erratic and cannot be trusted. Prompt injection attacks mean that llm responses are generally untrustworthy. Even without prompt injection, agents can still mis-use functions. Programs need to be minimally scoped and agent design must assume nefarious usage. Secrets cannot be accessible to the llm. For example, if a program wants to execute code from the agent, it should use a separate environment to execute the code to prevent secrets from leaking. Memory write/delete should be limited to the scope of the conversation. Modification via dreaming needs to undergo QA and be easily reversible.

We draw our security boundary at the machine level. This means a malicious program has access to all the resources of anything on its machine. Permissions and secrets need to be treated appropriately. To silo programs or groups of programs, create two separate machines to run them. Since the communication interface is identical between machines and programs, this should be a change in architecture more than a logical change. Similarly, memory between machines is not shared.

Since some agents will be system aware, deployments will need to enforcing crud permissions similar to kubernetes[4]. This will likely be deployment specific and may be beyond the scope of the open source framework.

The agent framework will provide programs with pluggable memory. This provides agents with another avenue of communication/collaboration, but also needs to be siloed if desired.

Agent Versioning

Program and machine specifications will have a version which conforms to traditional semantic versioning. Users will be responsible for maintaining this in their own CI systems. This should rev on version changes (new runtime image, different configuration, etc). Interestingly, as agents

[4] We can leverage existing deployment environments to manage this. We don't want to be yet another kind of permission to manage within a deployment. For example, when deploying in k8, we will want to create a custom "agent" resource and an agent controller to manage state. Serverless deployments would need a similar controller so be able to enforce CRUD permissions.

learn in deployed systems (ie, memories or insights), they may eventually change sufficiently to warrant a new version of the agent or something similar.

For example, a tool which derives insights from a conversation to use for future conversations will likely not need to manage versioning, but it needs to live alongside capabilities to view, or remove these insights. A dreaming tool or agent will need to provide hooks to integrate with CI to modify the agent spec and version.

Restore Backup

Agent restoration consists of more than just the agent code/configuration at a particular time. It also may include experiences up to a given point and time. In addition to being able to revert the spec and redeploy an agent (easily managed by version control), behavior altering tools need hooks to rollback state to a given point in time. The framework will provide hooks for this, but the onus becomes on the tool developer to implement these features. The framework can provide functionality to make this easier (such as automatically storing versions in memory records).

Agent Evaluation

A/B Testing

Promotion

Inter-agent Communication (Contract between agent container and deployment env)

Deployment Environment

Agent Management

Monitoring

Logging