On-Demand Intelligent Agent Generation

Brian Stensrud, Asher Gibson, Sean King
CAE USA
Orlando, FL
{brian.stensrud, asher.gibson, sean.king}@caemilusa.com

ABSTRACT

With the imminent emergence of peer adversaries, the DoD must elevate its design and use of simulation-based training exercises so that they better represent the complexities of anticipated future conflict. It is widely understood that future conflicts will be multi-domain and will require simulating hundreds of disparate actions and behaviors. To model these behaviors in the absence of human role-players, we rely on computer-generated forces (CGFs) – long a staple in constructive and virtual simulation environments. CGFs can be used to simulate teammates, OPFOR, and pattern-of-life behaviors that all contribute to realistic training scenarios (Stensrud et al, 2012).

While there is no shortage of techniques, architectures, and authoring tools for building and integrating these CGF agents, the time and expertise required to develop those behaviors is often cost-prohibitive. Large Language Models (LLMs) provide a compelling alternative to manual CGF behavior development and authoring. Optimally, a robust LLM – tuned with knowledge about the target domain - could generate intelligent agents directly from prompt, saving hours of manual knowledge engineering and behavior development. Unfortunately, such an approach ignores essential steps in the process, such as validation, debug, and integration with a target simulation.

We introduce a novel architecture for generating intelligent agents for CGFs that situates custom-tuned LLMs into an agentic framework capable of composing, refining, evaluating, and integrating new behaviors into a specified simulation environment from prompt. We discuss the process involved iterating on the design of this architecture, the limitations discovered in targeting specific behavior representations. As simulations can vary drastically in how they represent and control CGF behaviors, this architecture includes an approach for automatic discovery of actions and state-retrieval functions, allowing the generated agents to operate seamlessly within various simulation environments.

ABOUT THE AUTHORS

Dr. Brian Stensrud is currently serving as CAE's Technical Fellow for Artificial Intelligence, providing guidance and strategy on the adoption, application, and design of AI solutions across both its Defense and Civil Aviation business units. Brian has over 20 years' experience designing, developing, and deploying AI-enabled training and decision-support systems, and has led the execution of over 30 DoD-sponsored RDT&E efforts across many of the services including ARL, ONR, AFRL, and DARPA. Brian received his Ph.D. in Artificial Intelligence from the University of Central Florida in 2005, and holds undergraduate degrees in Computer Engineering, Electrical Engineering, and Mathematics from the University of Florida.

Asher Gibson is currently working as a Machine Learning Engineer Co-op at CAE USA, helping to create solutions for using locally hosted large language models for secure and efficient tasks requiring complex inference and constrained generation. Asher has gained over two years of experience in the defense industry in his time at CAE and has helped to integrate complex systems consisting of over 20 individual services. Asher has an undergraduate degree in Computer Science from the University of Central Florida and is currently pursuing a graduate degree in the same field.

Sean King is a Machine Learning Engineer and lead developer in CAE USA's artificial intelligence group.

On-Demand Intelligent Agent Generation

Brian Stensrud, Asher Gibson, Sean King
CAE USA
Orlando, FL
brian.stensrud, asher.gibson, sean.king}@caemilusa.com

INTRODUCTION

Intelligent agents are a staple of constructive and virtual simulation environments. They can be used to serve as teammates, OPFOR, or non-kinetic entities such as pattern-of-life behaviors that contribute to realistic environments—particularly in urban areas. As such, there is no shortage of techniques, architectures, or even authoring tools for building and integrating these agents, and many of these techniques are sufficient for modeling complex supporting tasks and behaviors.

A key drawback, however, is the time and expertise required to develop those behaviors using these techniques. For example, cognitive architectures (Laird, 2019; Ritter et al, 2019) are powerful engines capable of representing and executing complex agent behaviors, but they require significant technical expertise to use for developing agents. Even simpler representations, like finite-state machines or behavior trees, require extensive work to design and implement robust behaviors. In many cases, the behavior developer building out the agent typically must learn a great deal about the behavior he/she is modeling—from field manuals, subject matter expert interviews, followed by endless testing, debugging, and experimentation. The time, cost, and expertise to develop agents in this way simply does not scale to the numbers of behaviors necessary for large-scale exercises.

Recent advances in GPU (graphics processing unit) technology, along with new generative artificial intelligence (AI) techniques emerging from industry (including transformer models, deep reinforcement learning, and others) provide an opportunity to rapidly develop and integrate robust intelligent agent models. Over the past year, the authors have been prototyping a capability for rapidly generating behavior models – an *agent factory* - that can integrate and operate in constructive simulation environments.

CONCEPT

We seek to employ a large language model to expertly generate dynamic text-based representations of agents—autonomous controllers of entities within scenarios of a computer-generated forces (CGF) simulator—based on textual descriptions of the responsibilities of those agents within the desired scenario-to-be-created and the key events that define that scenario. We want the output generated by the language model to be converted directly into usable and runnable artifacts that can represent the intricate relationships, definitions, and patterns of autonomous warfighters in scenarios involving complex joint all-domain forces.

The central concept of our agent factory approach is to enable simulation operators—particularly battlemaster personnel—to dynamically generate intelligent behavior models for computer-generated forces within constructive and virtual simulation environments. Rather than relying on pre-authored, static behavior models that are often costly to develop and difficult to adapt, our architecture empowers users to create tailored agent behaviors on demand, using natural language specifications.

At the core of this capability is a large language model (LLM; Vaswani et al, 2017) that interprets user-authored behavior specifications and translates them into executable agent logic. These specifications can range from simple task directives (e.g., “follow route CHARLIE and report OPFOR contact”) to complex operational roles involving rules of engagement, conditional responses, and reporting protocols. To ensure domain relevance and accuracy, the LLM can be augmented with doctrinal materials—such as field manuals or training guides—via retrieval-augmented generation (Gao et al, 2023) or fine-tuning, depending on the availability and structure of the source material.

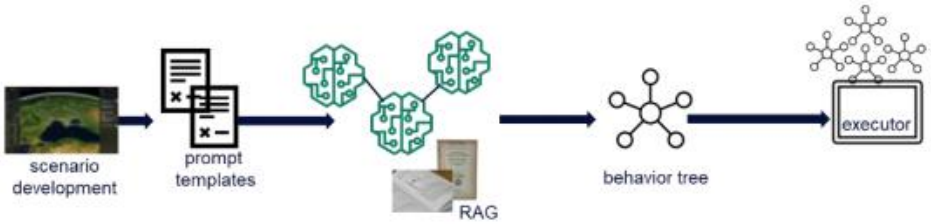

Figure 1. Conceptual flow of Agent Factory

Initially, we explored generating agent behaviors in the form of behavior trees—symbolic, XML-based representations that are both interpretable and compatible with many simulation engines (see http://behaviortree.cpp). Behavior trees offer a clear advantage in terms of traceability and validation, allowing developers and operators to inspect and debug agent logic prior to execution. This symbolic representation also avoids the performance penalties and opacity associated with sub-symbolic approaches, such as deep reinforcement learning or direct LLM inference at runtime.

One alternative we considered was using a large language model as the behavior engine itself—feeding it the current simulation state as a prompt and having it return an action at each simulation tick. While conceptually elegant, this approach presents two critical drawbacks. First, the computational cost of running inference on an LLM at every tick cycle becomes prohibitive, especially at the scale required for large training exercises involving hundreds or thousands of agents. Second, the behavior generated by such a model would be sub-symbolic and inherently opaque. Without a symbolic representation of logic, it becomes nearly impossible to validate, debug, or explain why an agent chose a particular action—an unacceptable limitation in training environments where traceability and accountability are paramount. By contrast, the approach we settled on shifts inference to pre-runtime and produces symbolic, human-readable behavior logic, enabling both performance efficiency and operational transparency.

We encountered limitations in expressiveness and flexibility when using behavior trees, particularly for more nuanced or dynamic behaviors. As a result, we transitioned to generating agent logic directly in Python code. This approach retains the benefits of symbolic representation—transparency, explainability, and pre-runtime inference—while offering greater flexibility in behavior composition and integration with diverse simulation environments.

A key innovation in our architecture is the concept of "agents as disposable commodities." Traditional agent development emphasizes reusability, often resulting in tightly coupled behaviors that are difficult to adapt across different scenarios or simulation platforms. In contrast, our approach embraces disposability: agents are generated rapidly and specifically for the scenario at hand. This paradigm shift is enabled by the speed and ease of behavior generation using LLMs, allowing operators to create scenario-specific agents without the overhead of maintaining a library of reusable behaviors. If a scenario changes, new agents can be generated just as quickly, ensuring alignment with updated objectives and environmental conditions.

To support integration with a wide range of simulation environments, our architecture enables mechanisms for automatic discovery of available actions and state-retrieval functions. This allows generated agents to interface seamlessly with the simulation's API, regardless of how CGF behaviors are represented or controlled. The result is a flexible, scalable, and operator-friendly system for generating intelligent agents that meet the demands of modern, multi-domain training exercises.

METHODOLOGY

We decompose here the agent factory workflow into four logical steps, listed and described below:

1. **Foundation Model Selection:** Choosing a foundation model capable of generating accurate, viable Python code that satisfices the user's specification as prompted
2. **Agent Generation and Validation:** Engineering the appropriate prompt templates, generating the agent model, and modifying for correctness if necessary
3. **Simulation Integration:** Instantiating the generated model into a simulation environment, and connecting it to a target CGF entity
4. **Execution:** running the agents to stimulate CGF entities within a scenario

We adopt the following terminology to describe the various components at play in this workflow:

- *Agent:* an autonomous control algorithm or controller for an entity behavior in a simulation
- *Agent Definition:* the executable code generated by the agent factory, that defines the agent's behavior
- *Entity*: the air, ground, naval, or other force; represented in simulation, that is controlled by an agent
- *State:* the set of environmental variables and conditions the agent uses to make decisions
- *Action:* what the entity does, each simulation tick, in response to its mission and the current state

Foundation Model Selection

Selecting the right foundation model for agent generation proved to be less about published benchmarks and more about empirical performance in context. While many large language models (LLMs) boast impressive statistics, we found that these metrics often fail to translate into meaningful differences when applied to our specific use case. Instead, we adopted a pragmatic approach: run the models, compare outputs, and choose the one that consistently delivered the most coherent and accurate agent behaviors.

Our initial exploration included LLaMA 3, Qwen, and others, but it was Phi-4 Mini that performed best. Phi-4 was initially considered based on claims that it would serve as the backbone for Copilot and that it could match the performance of models 3–4 times its size. While we cannot independently verify those claims in full, our testing supported the notion that Phi-4 offered comparable—if not superior—performance in generating agent logic, particularly in terms of accuracy and coherence. Its smaller footprint also made it an attractive option for local deployment and rapid iteration.

Phi-4 Mini became our default model for most LLM-based agent generation tasks. Importantly, we evaluated its capabilities using minimal configuration: a temperature setting of 0.2 (the lower bound of the recommended range) and no additional fine-tuning or system prompt constraints. This allowed us to assess the model's out-of-the-box performance and robustness without relying on extensive prompt engineering or hyperparameter tuning. The results were consistently strong, reinforcing our decision to work with Phi-4 for the factory.

Agent Generation and Validation

Generating effective agents – or any substantive material out of LLMs - begins with prompting, the process of crafting structured input that guides the language model to produce the desired output. As LLMs are highly sensitive to how tasks are framed, prompt engineering allows us to iteratively guide the model to produce effective responses. In our workflow, prompts define the agent's role, available actions, and expected behaviors, serving as the blueprint from which the model constructs executable logic.

Our prompt engineering strategy includes two tenets: First, the fact that state-of-the-art model training already incorporates some popular tactics such as chain-of-prompt reasoning, which encourages the LLM to generate intermediate reasoning steps before arriving at a final output (Wang et al, 2024). This means that acceptable performance can be achieved with simple strategies like zero-shot or one-shot prompting (Pourpanah et al, 2022). The system prompt contains the available actions that the agent could perform. These available actions are included as Python function headers, which include the types of the expected parameters and a brief description of what the function does. The system prompt also contains instructions to add a TODO marker to any action the LLM is unable to write, should a needed action be missing from the list of available actions. The template of the agent design pattern was also included. It was decided that to make validation as simple as possible, to confine the language model to following a predetermined pattern specifying where variables should go, where conditionals should be defined, and where default behaviors should be executed. A single example of a very simple agent is provided. This was decided to prevent the LLM from getting too much assistance from the example given, since the long-term vision is to evaluate how robust the LLM is when asked to create more complex agents.

The second tenet of our prompt engineering strategy is how the user should prompt the agent. Specifically, our requirement is that agents need to be reactive to their environment; not merely execute a single script. The LLM was prompted in a structure that starts with "Create a <role> agent" where we define what the agent is going to be (i.e. fighter, tanker, etc.). The user prompt then contains prescriptions for what the agent should do if certain conditions

are met. For example, "if the agent detects a red force within 20 km of itself, the agent should approach the red force." We itemize the conditions and the desired behaviors in a list format:

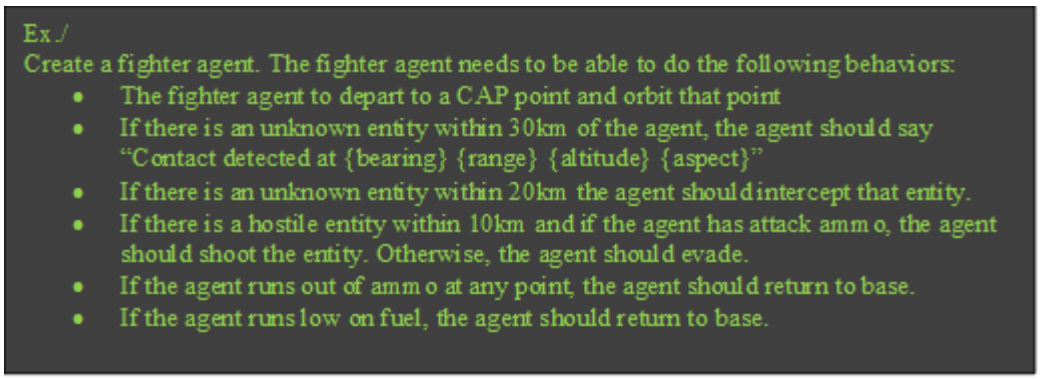

Figure 2.A sample prompt used to create a runnable agent of the appropriate, defined text-based structure.

For validating the correctness of the LLM output, a human manually inspects the code for bugs, syntactic and semantic. Syntactic bugs were uncommon, but semantic bugs could occur due to careless wording in the behavior prescriptions. After inspecting the agent for logical correctness, the agent was then loaded into the agent runner interface with the CGF to see if the behaviors were performing as expected. Any changes or tweaks were then made by the validating human and then re-executed in the CGF until the desired behaviors were observed.

Generated agents were manually graded to assess their completeness and functionality – we tabulate the results of that grading before. In the table, the *syntactic score* represents whether the agent can be run or not, as agents containing obvious errors cannot be run to accurately assess semantic score. (As such, generated agents with a syntactic grade of 0 were also given a semantic grade of 0.) The *semantic score* is the percentage of the actions and conditions from the prompt that the resultant agent could perform correctly, according to the human grader.

Table 1. Average scores for all agents generated. Prompt type is determined by the details of the prompt used for the factory. A 'Procedure' is a prompt for a behavior containing a sequence of events the agent must perform in correct order. The 'Other' category is used for prompts involving agents that do not necessarily follow a sequence of events.

Prompt Type	**# Generated Samples**	**Avg. Syntactic Grade**	**Avg. Semantic Grade**
Procedure	8	0.875	0.756
Other	2	1.000	0.925

Table 2. Agents generated by procedure descriptions. The average scores for agents generated using descriptions of procedures are displayed above. Each procedure was categorized according to the task requested in the prompt. For each agent, a score between 0 and 1 is given for the syntactic category and the semantic category. For an agent with a single syntactic mistake, a score of 0 was assigned. Agents without any syntactic errors were given a score of 1. For semantic errors, a scoring guideline was followed, as described below.

Category	**Avg. Syntactic Grade**	**Avg. Semantic Grade**
Detection	1.000	0.909
Routing (distance-based)	1.000	0.788
Routing (time-based)	0.667	0.619
Other	1.000	0.920
Average	**0.875**	**0.756**

Connecting Agents to the Simulation

Once a valid agent is generated, it needs to be connected to the simulation environment so that it can control an entity. This requires (1) linking the agent to its target entity within the environment, (3) exposing the set of simulation actions available to the entity, and (3) configuring the entity so that it can pull state data from the simulation.

Most of this process is handled at agent generation time through the prompt template. Entities within a simulation typically have a unique identifier (*uid*) and assigned type, which specifies how it can be referred to and what type of model it represents. As such, this information can be appended to the agent definition as variables and can thus be specified in the original prompt. For instance, Figure 2 provides prompt information for generating a simple agent behavior. To add information to bind that agent to an entity within the simulation, we could simply add the constraints to the prompt, like "*generate a constant named **uid,** and assign that uid the value **F16-001**. Generate another constant named **entity_type**, and assign entity_type the value **F_16.**"* The resultant *agent structure*, described in the following sections, retrieves that information at simulation run-time to make the connection.

Both state information (which agents require to build situation awareness and inform its decisions), as well as the specific actions available to take in the environment (and the parameters needed to pass to those actions), are often unique to the simulation. Even calling relative actions, like "*move to location,"* can look quite different from one simulation to another. This provides additional motivation for our agents as disposable commodities – since we can generate them on demand, it is acceptable to generate them in such a way that their actions and decision-making are tightly coupled to the actions and state available within the simulation in which they will be operating. As such, we can use the information provided to us by the simulation (e.g. defined in interface control documents or related documentation) and provide that within our prompt – listing out the set of all available actions, parameters to be passed to those actions, and the names/descriptions of state variables that the agent can use to reason and act. During execution, then, agents receive dynamic updates of these variables through ingestion of state traffic published over some protocol (e.g. DIS, HLA). This traffic consists of a constant stream of detailed information about the state of the simulation environment and the entities within it at scenario execution. Protocol Data Units (PDUs) are read from at each simulation tick to populate an internal "state" variable accessible to the agents within Python code. Conditions regarding values present within this state variable are used to guide agent action and behavior as the scenario plays.

Inspiration behind the Agent Structure

Reinforcement learning approaches to represent artificially intelligent agents within a scenario utilize an observation space, a policy, and an action space to employ a neural network as a 'next action' predictor. The *observation space* is used as input to inform the underlying neural network of the current state of the environment as the agent perceives it. The *action space* is the set (or range) of all possible choices the agent can make to perform actions within a scenario. A *policy* is used to help the model work towards long-term goals with its actions over time.

Our approach mirrors the observation-policy-action architecture used in reinforcement learning:

- The *policy* is the agent definition, generated by the LLM which defines how the agent will react when specific conditions arise in the scenario
- The *observation space* contains the state of the simulation environment perceived by the agent at a given point in time. This is represented as a key-value store which can be read and used in Python code (as opposed to a matrix input for neural networks)
- Our *action space* is the set the possible actions the agent can perform, but will be a human-readable action enumeration, string, or call with human-readable parameters

The LLM is responsible for writing the appropriate policy informed by the action space. We discuss the details of how this translation from LLM output to a functional agent which can be repeatedly invoked to obtain a 'next action' prediction occurs in the following section. As Python code is interpreted, agents do not require compilation with source code and can be exchanged easily with other agents as new use cases arise. Python also has support for many common programmatic structures, including a few advanced structures, such as *generator functions*.

Python generator functions have three unique properties that can be exploited in helping to represent readable behavior. Generator functions can 'yield' values temporarily, returning execution to the caller (Property #1). The values returned can be interpreted as the outputs of those generator functions. Generator functions can also be transformed into an iterator of indeterminate length (Property #2), as length is determined by the number of times 'yield' statements are called before reaching the end of the function, where the end of the function is known to the caller via a 'StopIteration' exception. The final property manifests such that when generators are called, a call to a special inbuilt function called '__next__()' is invoked on that generator. Because of this '__next__' function,

execution within the generator function starts exactly where it left off, enabling the generator to be 'stateful' (Property #3).

In Figure 3, we show a sample agent structure for an agent for an A-10 Warthog that can, upon perceiving environmental stimuli, perform sequences of simple actions that, in practice, are replaced by actual doctrine-based approaches generated by a language model. This is an example of how the logic behind performing doctrine can be compressed into an automatically generated, runnable text-based data representation that is simple, easily reproducible, human readable, and generally applicable.

```
# Our agent action generator
def agent():
    # Define the state object
    global state

    # Define constants
    MY_NAME = "A-10 Warthog X"
    THRESHOLD_RADIUS = 3.0

    # While the agent is alive
    while state[MY_NAME]["health"] > 0:

        if cond1 ... :
            ...
        elif cond2 ... :
            ...
        elif cond3 ... :
            ...

        ...

        elif state[MY_NAME]["closest-threat-distance"] < THRESHOLD_RADIUS:

            # Obtain environmental information
            enemy = state[MY_NAME]["closest-threat"]

            # Potential doctrine-based approach and procedure

            yield (Action.FLY_TOWARDS, {"target": enemy})

            yield (Action.FOLLOW, {"target": enemy, "offset": 1500})

            yield (Action.SHOOT, {"target": enemy, "weapon": "GAU-8/A 30mm"})

        ...

        else:

            # Do as commanded
            yield (Action.CURRENT_COMMAND, {})
```

Figure 3: A generated agent structure

When inputs to the generator function are the environment state and outputs are actions to respond to the given state, this generator function becomes logically equivalent to a 'next action' predictor from an external perspective. Logic for how to internally handle state is kept completely inside the generator function, allowing custom implementations of doctrinal procedures to lie within the agent's action generator function based on that agent's type within the environment. For example, an F-16 agent would act significantly differently when approached by a hostile aircraft than a B-2 bomber agent, so this internal implementation for action prediction based on state would be different for the two entities. Upon yielding values, which get parsed into actions, execution is returned to the caller, allowing the external runner thread to perform the necessary code needed to run the associated output action within the environment, enabling environment-agnostic application for the action generator. When called again, the generator picks up where it left off, enabling a sort of "memory" of what action was performed and what state that agent was in when execution of that agent resumes. This also enables the generator agent structure to fit seamlessly within the initial restrictions and environment of the behavior tree agents, where a call to '__next__' on a generator directly replaces the tick of a behavior tree.

One limitation of this approach is that the state input object has to be a global in order to be updated outside of the agent generator (with any stimuli or changes from the runtime environment) and simultaneously have those external updates reflect within the state object inside of the generator.

Running the Agents

Agents generated with this structure align well with a tick-based simulation architecture - the most common architecture adopted by major modern CGFs. In our investigation, this tick-based execution enabled the language model-generated agents to act as a drop-in replacement to the behavior tree-based agents from our initial implementation.

Pseudocode for the simplest possible reproduction of the agent runner is shown in Figure 4. First, the state object representing a lookup table of all possible entities (and their individual states and properties at the current simulation time) and all environmental factors and conditions within the simulation is declared and instantiated. As shown, the runner takes a set of *N* agents, each represented by the text-based multi-line string Python generator definition that can be output by the language model, as input. *N* is equal to the number of agents that are in the scenario at its start. These string definitions are interpreted via a call to 'exec', creating the associated generator functions within the Python instance. The generators are then transformed into their corresponding iterator objects to ensure the agent generators are automatically updated when '__next__' is called.

After this, the simulation is run, and each iteration within the main simulation loop is a 'tick' to the entire simulation. The agents are ticked sequentially, with a '__next__' invocation to receive the action the agent would perform based on the current state. After the agent returns this value, a call to perform that action within the environment is made. After all agents have been ticked once, the state is ticked and updated with new information.

```
# Declare the state global
global state
# Create the state global
state = instantiate_state()

# Declare the generators of all agents in the simulation
generators = [exec(agent_string) for agent_string in agent_strings]

# Transform the generators into iterator objects
agents = [iter(generator) for generator in generators]

# While the simulation is running
while state.simulation_running():

    # Go through all agents
    for agent in agents:

        # Call '__next__()' to 'tick' the agent and obtain the action
        action = agent.__next__()

        # Make the agent perform the action in the simulation
        state.perform(agent, action)

    # Update the simulation environment
    state = update_state(state)
```

Figure 4: A simple runner for a simulation of agents generated using our method.

Agent Reusability and Customization

Although agents can be created automatically and discarded (our 'disposable agent' paradigm), agents can also be made in ways that can generalize and apply to more than one scenario. Agents can be created to be generic and human-like, representing what a human would do while following doctrine in situations, whether at peace or at war.

However, many military training scenarios that agents can be used within to represent ground, naval, or air forces involve detailed or exceptional circumstances and mission-critical deviations to properly prepare warfighters for how to handle future, unexpected events. If we take an example of two agents from two different sides approaching one another, and one of the agents fires a missile at the other, this would be a deviation from standard doctrine if the two sides are at peace and a war is starting immediately within the scenario.

Therefore, for the case of reusable agents, we need a concept of a *timeline* for any given scenario that may involve abnormal events to account for and recreate abnormalities. On this timeline are actions that may alter the scenario significantly and drive the surrounding agents to act differently. These timeline actions are issued to a single agent and force that agent to perform that action at the specified time during the scenario's execution (i.e., forcing one of the agents to fire the first missile despite the two sides being at peace).

The existence of an added service to send these mission-driving deviations enables generic agent definitions to support additional possible scenarios, even if the mission requires actions or commands that would not be orthodox according to standard doctrine.

APPLICATION

The agent factory is currently in use as part of an overarching series of LVC demonstrations[1] supported by the Pacific Multi-Domain Training and Experimentation Capability (PMTEC) Program. PMTEC is an organization within the USINDOPACOM J7 Training and Exercises Directorate; Its charter is to deploy forward joint training capabilities in the INDOPACOM Area of Responsibility (AOR), particularly those areas West of the International Dateline referred to as the Second Island Chain (SIC), and possibly the First Island Chain (FIC). PMTEC's technology infusions enable joint, combined, and coalition warfighters to realistically rehearse operations in highly contested, all-domain environments. Per the J7 Director, this is crucial for preparing against peer adversary capabilities and supporting integrated deterrence. Among the training capabilities PMTEC initially brought forward were largely existing and stand-alone LVC training systems; When these capabilities were integrated under PMTEC's visionary oversight, they formed the basis for a future Advanced Training Environment (ATE) to create a more realistic presentation for mission rehearsal. These efforts were furthered during Valiant Shield 2024 and Sling Stone 24 Capabilities Exercise (CAPEX), held on or about Guam; PMTEC partnered with industry and DoD stakeholders to derive true needs and

[1] *See I/ITSEC Paper #25350 for additional information about these demonstrations*

deliver C/JLVC capabilities across the Pacific theater to allow joint and coalition warfighters to conduct high-end training without revealing tactics, techniques, and procedures to our adversaries.

Over the past year, PMTEC has sponsored multiple joint LVC "operational readiness exercises" including Valiant Shield 2024 (https://ipdefenseforum.com/2024/07/expanded-valiant-shield-2024-highlights-advanced-warfighting-capabilities-in-indo-pacific/), Sling Stone 2024 (https://www.cpf.navy.mil/Newsroom/News/Article/3994074/sling-stone-enhances-warfighter-capabilities-defense-of-guam/), and Bamboo Eagle (https://www.nellis.af.mil/News/Article-Display/Article/4060640/us-allied-air-forces-and-joint-partners-strengthen-readiness-in-bamboo-eagle-25/). In these exercises, the LVC components included:

- **Live**: Blue Force Training Audience using operational hardware/software – (e.g., live flight aircraft, Army Integrated Battle Command System (IBCS) for Terminal High Altitude Area Defense (THAAD) and Patriot Batteries, Air Force Tactical Operations Center – Light (TOC-L))
- **Virtual**: Blue Force Training Audience using dedicated platform simulators (e.g., Virtual Aegis Ashore Weapon System (VAWS), Navy Cooperative Engagement Capability (CEC), fast-jet fighter desktop trainers, ISR feeds)
- **Constructive**: Red force entities and Blue Force entities – as required to account for where there were no live Blue Forces (e.g., C-RAM Distributed System of Systems Simulation (CDS3), various commercial-off-the-shelf CGFs)

The agent factory is being used to generate constructive entities for both the red and blue side to support the exercise. Specifically, these entities are generated so that they can provide intelligent clutter for the scenario, executing peripheral (but relevant missions). To make this happen, first a set of vignettes are conceived and generated manually by planning personnel (e.g. mission objectives, initial force laydown, rules of engagement). These vignettes will include identification of what entities need to be instantiated, and what their actions/behaviors need to be. That information is used to generate a set of prompts (see Figure 2), which cue the generation of behaviors for each entity.

CONCLUSION AND FUTURE WORK

The Agent Factory represents a significant step forward in the rapid generation of intelligent agents for constructive simulation environments. By leveraging large language models to produce executable, interpretable agent logic from natural language prompts, we have reduced the time, cost, and expertise traditionally required for agent development. Our architecture emphasizes disposability, flexibility, and integration, enabling simulation operators to generate scenario-specific behaviors on demand. A critical component of this workflow was the validation, integration, and execution of generated agents. Each agent underwent manual inspection for logical correctness, followed by integration into a simulation environment where its behavior could be observed and refined. This iterative process ensured that the agents not only compiled correctly but also performed as intended within dynamic, tick-based simulations—bridging the gap between language model output and operational utility.

While the agent factory concept has shown immense potential, and has already demonstrated utility for current LVC exercises, significant work remains to make it a fully robust and scalable solution. As we describe in the paper, several stages of the workflow—such as prompt authoring, agent validation, debugging, and simulation integration—still require human oversight and iterative refinement. These manual touchpoints limit throughput and introduce variability in agent quality. As we continue development, we will be targeting opportunities where these steps can be automated or tightly integrated into the broader scenario generation process. Embedding validation logic, improving prompt templating, and enabling automated feedback loops will be essential to reducing human-in-the-loop requirements and ensuring consistent, high-quality agent behavior across diverse simulation contexts.

One approach we are considering to make the agent factory accessible to a larger user group is to develop an interface where prompting within the formal prompt template, and querying the language model to generate custom agents, can be created. This interface should consider the underlying complexities of agents generated and expose this generation in a way that is amenable to a non-technical user with limited domain-specific knowledge. Democratizing the creation of agents is an important step in making the creation of scenarios within computer generated force simulators less time-intensive and less challenging.
A second step in the current process that is manually intensive is the definition of simulation action and state space. Agents rely upon a discrete list of available actions directly within the prompt structure that inform the LLM as to

what a generated agent can include, call, yield, or output to interact with the simulation. Automatic discovery and placement of this set of actions within the prompt can be implemented through an additional step which would involve ingestion of CGF or CGF plugin documentation or code via an LLM which could parse and output the set of available actions. This could expand the applicability of the factory's generated agents to a greater number of CGFs and simulation environments.

Validation of generated agent logic is also a manual step in the process. Through repeated trial and error via running the newly generated agent in the simulation software and observing the controlled entity's behavior, the agent definition is adjusted to ensure all conditions, parameters, and associated actions correctly embody the original description given to the factory. In the future, this process can be automated, using an LLM-based testing framework which can perform corrections to the agent definition automatically. A system can be implemented to run the agent, record the agent's behavior over time in video or log-based textual format, and make the necessary adjustments until the agent's behavior exactly matches the description.

Currently, the Agent Factory is designed to generate individual agents based on generalized task descriptions, with each agent developed independently of the broader scenario context. While this enables rapid creation of doctrinally sound behaviors, it also means that additional customization is often required to ensure agents behave appropriately within the specific dynamics of a given scenario. As the system matures, a natural extension would be to support coordinated multi-agent scenario generation, where agent behaviors are informed by a shared understanding of the scenario's structure, timeline, and inter-agent dependencies. This would involve two key enhancements: (1) parsing a high-level scenario description—including force composition, key events, and environmental context—to identify the roles and responsibilities of each agent, and (2) using those role-specific expectations to guide the generation of each agent's behavior via tailored prompts. Ultimately, this would enable end-to-end scenario generation from a single textual or spoken description, reducing the need for manual alignment and post-hoc adjustments.

Finally, in the future we seek to explore the development of standardized benchmarks and evaluation protocols tailored specifically to agent generation tasks. These could include metrics for code correctness, behavioral fidelity, interpretability, and integration success within simulation environments. Additionally, automated tooling could be developed to run candidate models through a battery of representative prompts and scenarios, scoring their outputs against human-validated ground truth or expert heuristics. This remains an open research challenge: how to evaluate LLMs not just on general language tasks, but on their ability to generate structured, executable, and contextually appropriate agent logic. Addressing this gap will be critical to ensuring consistent performance and reliability as the agent factory architecture matures.

REFERENCES

Stensrud, B., Purcel, E., Fragomeni, G., Woods, A., Wintermute, S., and Garrity, P. (2012), "No More Zombies! High-Fidelity Character Autonomy for Virtual Small-Unit Training," Proceedings of the Interservice/Industry Training, Simulation and Education Conference (I/ITSEC) 2012, Orlando, FL, December 3-6, 2012.

Laird, J. E. (2019). *The Soar cognitive architecture*. MIT press.

Ritter, F. E., Tehranchi, F., & Oury, J. D. (2019). ACT-R: A cognitive architecture for modeling cognition. *Wiley Interdisciplinary Reviews: Cognitive Science*, *10*(3), e1488.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., & Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, *30*.

Gao, Y., Xiong, Y., Gao, X., Jia, K., Pan, J., Bi, Y., ... & Wang, H. (2023). Retrieval-augmented generation for large language models: A survey. *arXiv preprint arXiv:2312.10997*, *2*(1).

Wang, X., & Zhou, D. (2024). Chain-of-thought reasoning without prompting. *arXiv preprint arXiv:2402.10200*.

Pourpanah, F., Abdar, M., Luo, Y., Zhou, X., Wang, R., Lim, C. P., ... & Wu, Q. J. (2022). A review of generalized zero-shot learning methods. *IEEE transactions on pattern analysis and machine intelligence*, *45*(4), 4051-4070.