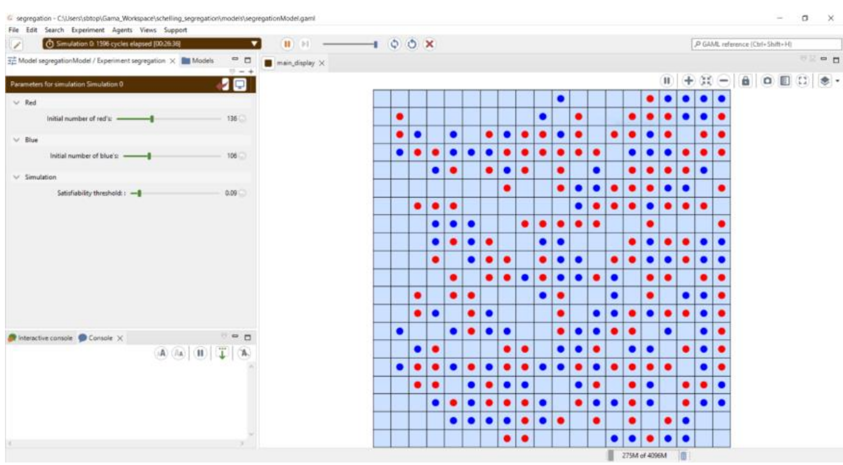
Streamlining EABSS with Generative AI: Part 2

I have spent some time to try and improve upon the approach to "Streamlining EABSS with Generative AI".

The virtual machine I used was the same as last time - a Google Cloud n1-high-mem machine with 26GB RAM and 4vCPU cores. I attached a T4 GPU to my virtual machine and running LLMs can now process input tokens and generate responses almost instantaneously, with similar responsiveness to ChatGPT. Before, running the EABSS script would take hours (if the SSH connection to the VM didn't timeout), now it takes 40-45 minutes (with an extended implementation). If you are running these tasks regularly, I would consider getting a GPU for a local machine to be a worthwhile investment, and nevertheless, having a GPU attached to a cloud VM is also an option.

Creating More Testing Models

I discovered an implementation of Schelling's segregation model, which I translated into GAML. This work was based on the information provided at http://nifty.stanford.edu/2014/mccown-schelling-model-segregation/. A screenshot captured during the experiment can be found below.



I also implemented a Bass-diffusion GAML model (please note: the experiment will probably need slowing down for careful inspection).

Furthermore, I developed a basic version of the adaptive museum model in GAML. The model contains visitors that can enter/exit a room along with a natural tendency to follow digital screens, which move randomly in a horizontal fashion. If I have time in the future, I might improve upon it and create a second version. Despite its simplicity, it remains quite an advanced model for LLMs because of the chain of thought needed to implement the logic for visitors to "follow" screens.

In addition to the models above, I implemented Conway's Game of Life in GAMA/GAML, another reference model for testing. This model serves as a widely understood and powerful benchmark for evaluating the capabilities of LLMs, due to its emergent behaviours that arise from simple rules when correctly implemented.

Automating the EABSS script (for the Ollama server)

I wrote a Python script to automate the EABSS script in the Ollama CLI. All prompts are stored in a JSON file, each prompt is stored in its own map, where the key is a unique name assigned to the prompt (e.g., the first prompt in the EABSS script is named "intro_1"), and the value is the text of the prompt itself. The script works by getting the necessary EABSS input (key-topic, key-researchDesign, key-domain and key-specialisation) from the user on startup. The script then automatically follows the EABSS script by injecting the user's input strings into predefined string injection markers contained within a JSON file. The conversation history between the bot and the LLM is printed to the terminal, and a copy of the conversation is also exported to a JSON file that contains a timestamp and the name of the model. This architecture makes it easy to edit prompts, add new ones and allows for quick testing. I have included instructions to use the script in the appendix.

Refining the EABSS Script

With the refined script, I aim to:

- Ensure that all important details/keys remain in memory for as long as possible (better key authenticity).
- Generate relevant and comprehensive UML diagrams using Mermaid.js, as much as possible.
- Generate more comprehensive GAML code that is syntatically correct, as much as possible.
- Update the script to minimise response errors.
- Make the format of EABSS script responses easier to read.

The refined script can be found in this folder in PDF format (*Seners_Refined_EABSS_Script.pdf*) and JSON format (*refined_prompts_as_json.json)*.

Setup

For refining the EABSS script, I used Mistral NeMo (12B parameters) as a middle-step between the LLMs that are too large to run on-device (such as ChatGPT) and the very small language models that typically have a short context length and low parameter count that can limit the complexity of their answers. Specifically, I used the default version of the model from Ollama (which has Q_4_0 weight quantisation and a context length of 32k) (https://ollama.com/library/mistral-nemo). The model requires ~7GB of disk space and uses ~13GB RAM when running with a 32K context length.

The required RAM can be reduced further by performing KV quantisation. In transformer-based models, key-value (KV) caches store the intermediate results of the attention mechanism for a sequence of tokens, allowing for efficient inference by avoiding recomputing attention scores for previous tokens. Ollama (and some other LLM servers) will configure the KV cache to store 16-bit floating point values by default, however smaller representations such as 8-bit floating points or even 4-bit integers can be used. More information can be found at https://smcleod.net/2024/12/bringing-k/v-context-quantisation-to-ollama/. I have added instructions to configure KV quantisation in Ollama in the appendix.

During testing, I found the default parameter settings given to Mistral NeMo models running on Ollama. If not stated in the MODELFILE, the temperature is set to 0.8, which does not align with the recommendations of the developers at Mistral and GPU infrastructure provider Scaleway, who recommend a temperature value of 0.4 (https://www.scaleway.com/en/docs/ai-data/managed-inference/reference-content/mistral-nemo-instruct-2407/). I also found a more comprehensive documentation of Ollama's default parameter settings (at https://docs.spring.io/spring-ai/reference/api/chat/ollama-chat.html). Code generation tasks demand high accuracy and adherence to syntax rules, a lower temperature helps achieve this by generating more deterministic outputs. I found that creative examples and interesting discussions (such as proposals for potential outputs) were still being generated when the temperature was set to 0.4. In the appendix I have included settings I used, along with how to configure them in Ollama.

Refining

Please note: the .txt files in the test folder are results of manual testing, while the .json files hold results of testing using the automation bot.

Before refining the script, I conducted some initial testing to assess the token usage for an average EABSS script. The testing revealed that the number of tokens required (both input and output) for an average script is approximately 20,000 to 22,000 tokens.

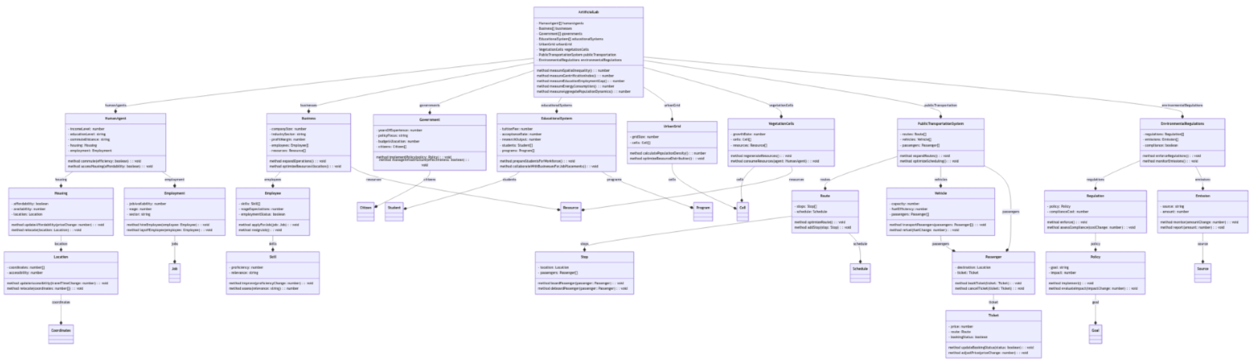
With early test runs of Mistral NeMo, I was getting syntax errors such as using the `function` keyword instead of `action` or `reflex` as well as missing type annotations for variables and attributes. To improve GAML code generation, I specifically asked the model about the different GAML syntax errors it recognises. Interestingly, a lot of the syntax errors that it seems aware of still get generated in EABSS script responses and sometimes, it tries to "correct" these syntax errors and ends up introducing new ones, the conversation can be found in *asking_mistral_nemo_common_gaml_errors.txt*. A possible explanation for these inconsistencies is the presence of conflicting or inconsistent examples in the model's training dataset. Moreover, the lack of high-quality GAML training data likely exacerbates this issue, although we can't be certain as we don't have access to the training data for such models. This behaviour highlights the need for additional guardrails and more robust code examples, which I have incorporated into the EABSS script (detailed later).

To overcome these issues, I decided to focus refining the script in two areas;

1. Refactoring UML design prompts (as these are used downstream for code generation)
2. Improving the implementation prompts by adding further "hints" to the LLMs, introducing code examples and better referencing prior steps.

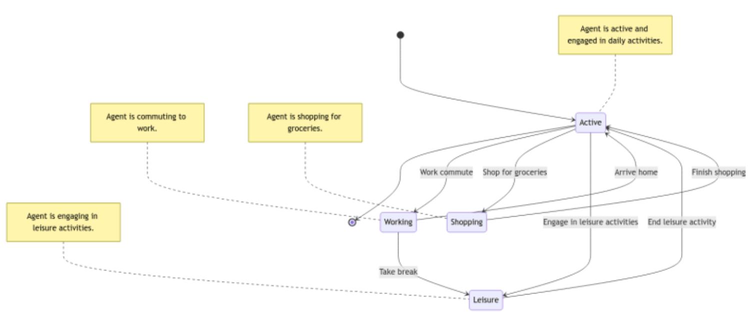
Refinement was an iterative process, and I tried the prompts across various use cases to ensure the model remained versatile – this is important as the EABSS script must remain generalisable across a variety of agent-based simulation scenarios. The use cases used for testing the refinements were Conway's Game of Life, flu epidemic, predator-prey and a market adoption model that used bass diffusion. The prompts used for each of these scenarios can be found in the *test_prompts* folder.

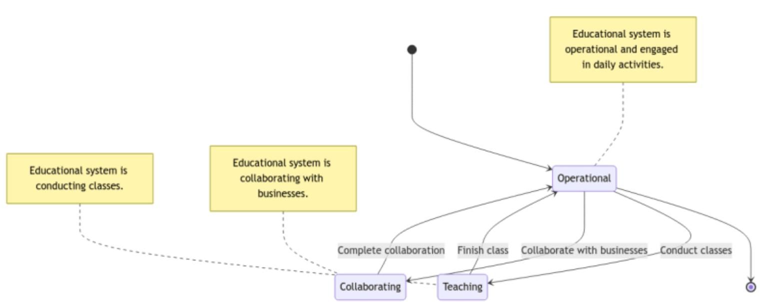
To refine the generation of class diagrams, I updated the script to explicitly begin with the `classDiagram` keyword. During early testing, alternative formats were occasionally used, even by larger models like Mistral-Small, which caused inconsistencies in the output. Furthermore, I restructured the instructions to ensure that the ArtificialLab class includes array attributes for each actor and physical environment object. This adjustment not only improved clarity but also addressed prior issues observed in test cases, such as the attached *predator_prey_test.txt*, where these elements were improperly implemented. Additionally, I introduced guidance on defining relationships between classes in the prompt that improves upon the draft class diagram. By specifying the use of `<|--` for inheritance, `*--` for composition, `o--` for aggregation, and `-->` for association, I ensured that the relationships between entities were both accurately represented and semantically meaningful. As evidenced in the tests timestamped on the 27th and 28th December, these refinements have significantly enhanced the comprehensiveness and accuracy of the generated diagrams.



Rendered class diagram from mistral-nemo_predator_prey_test_5_20241227-231107.json

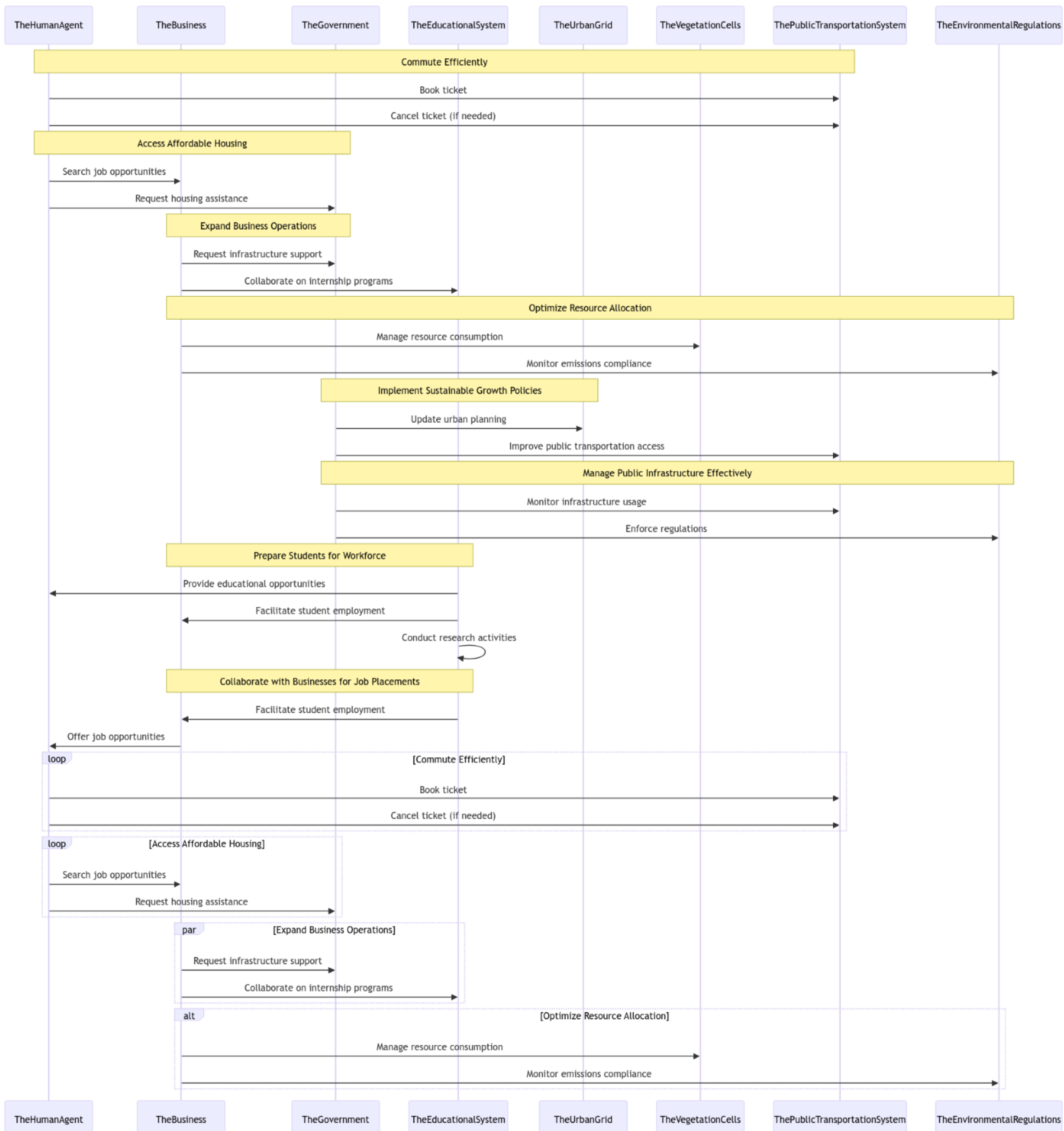
The generation of state diagrams underwent several refinements to enhance their precision and alignment with modern conventions. Firstly, I adjusted the script to explicitly request the use of the `stateDiagram-v2` notation to avoid issues stemming from outdated `stateDiagram` syntax. Additionally, I decided to explicitly instruct the model to generate separate scripts for each different element. I also removed some details from prompts such as asking the model to remove all state commands that include { and }, and I did not see the need to ask the model to replace all semicolons in the generated mermaid.js with full-stops – I never encountered this issue while testing. These refinements led to improvements as demonstrated in *mistral-nemo_flu_test_2_20241224-094001.json*, where the diagrams adhered closely to the prescribed format.





2 rendered state machine diagrams from mistral-nemo_predator_prey_test_5_20241227-231107.json

To improve the sequence diagrams, I provided explicit examples and instructions for labeling and structuring interactions. For instance, I incorporated a format for use case notes, specifying that they should be represented as note over <TheActorName>: <meaningful_note_here> along the vertical axis. Early tests, such as *mistral-nemo_predator_prey_test_2_20241222-180033.json*, lacked this clarity, but subsequent refinements addressed the issue effectively. Furthermore, I instructed the model to use ->> (solid lines) for interactions instead of --> (dotted lines), aligning with Mistral NeMo's preferences. This adjustment eliminated structural inconsistencies observed in earlier outputs. As a result, the test runs from 27th and 28th December (e.g. *mistral-nemo_predator_prey_test_5_20241227-231107.json*) present well-labeled, logically structured diagrams with properly defined interactions.



Rendered sequence diagram from mistral-nemo_predator_prey_test_5_20241227-231107.json (Mermaid.js correction: [parallel] keyword was replaced with [par])

For the first implementation step, I explicitly ask the model not to produce XML code as various language models have confused them in the past. I also specify the code is to be run in the GAMA engine, to provide more context and reinforce the fact that XML is not accepted. The phrase “COMPLETE, FULLY IMPLEMENTED” is included to prevent generation of boilerplate code. Leveraging context with the full conversation history is essential to provide the language model with a comprehensive understanding of prior discussions and goals. I make a point to mention the experiment block along with its contents as I have noticed it is one of the constructs most likely to be left out. I explain species in a way that hints towards using the correct syntax. All of this is reinforced with the scaffold at the end. The prompt also highlights class diagrams and sequence diagrams as references for structure and interaction between species to address the frequent mismanagement of actions and reflexes. By clearly describing actions and reflexes, the prompt mitigates errors in their application. Another critical issue, the improper initialization syntax (e.g., using <-), is explicitly addressed within the scaffold. The overall aim of the prompt is to establish a strong foundation—a draft script— that is refined in the subsequent prompts. Please find the prompt for the first implementation step below:

Using the information provided, I require a COMPLETE, FULLY IMPLEMENTED GAMA Markup Language (GAML) (NOT XML) simulation script for the key topic to run in the GAMA simulation engine. Use the context along with the rest of the conversation history. The GAML script must start with the "model" keyword, so the model starts with `model <APPROPRIATE_MODEL_NAME>`. Make sure to generate an "experiment {...}" block, with "output {...}" and "display {...}" blocks inside it. Use "species" to define actors/systems/species which you may have seen inside {key-mermaidClassDiagramScript}. Use interactions in {key-mermaidSequenceDiagramScript} to connect the respective species. DO NOT implement the artificial lab. To implement methods inside species, use `action` if the method is not expected to be executed at each timestep or `reflex` if the method is expected to be conducted at each timestep. Begin to generate the GAML script, remember it must be syntatically correct, valid (for example using `<-` to initialise all variables as well as attributes inside species blocks) and be FULLY IMPLEMENTED (all species, actions and reflexes). All interactions between species must be defined inside the appropriate species block. Use curly braces, rather than square brackets. Initialisation must take place in an "init {...}" block nested inside the "global {...} block". Make sure to refer to any “parameter” in the "experiment" block, if you think a modelling specialist would like to alter its value. Memorise this as {key-gamlScriptDraft}. an example gaml script scaffold is below:

```
model <gaml_model_name>
global {
     ...
     init {
```

```
            ...
      }
}
species <name> {
      <attribute_declaration_and_initialisation e.g. float probability
<- 0.5;>
      reflex <reflex_name> {
             <logic>
      }
      action <action_name>(<parameters_if_any>) {
              <logic>
      }
}
species <another_species> {
     ...
}

experiment {
      parameter <parameter_name>
var:<name_of_linked_attribute><parameter_options>;
      output {
           ...
      }
      display <display_1_name> {
           // if you want to visually show species
           species <species_to_visually_display> aspect: <aspect_name>;
           ...
      }
      // display other outputs separately (such as charts)
      display <display_2_name> {
           ...
      }
}
```

In a similar fashion to the UML diagram generation, I ask the model to improve upon its draft solution and produce a second draft. This prompt aims to connect the species and add detail to the GAML code to produce code blocks with substance–by commanding it: “`YOU MUST THINK CAREFULLY AND STEP BY STEP`”, as well as try to connect the species better. *mistral-nemo_conway_test_3_20241228-012645.json* demonstrates this –

all the methods generated in step 1 were stubbed, their actual implementations have been generated at this step. Please find the prompt for the second implementation step below:

```
Make sure every species, reflex and action in the memorised {key-
gamlScriptDraft} is fully implemented, YOU MUST THINK CAREFULLY AND
STEP BY STEP WHEN IMPLEMENTING THEM. For example if a species named
"species1" calls an action/reflex named "move()" belonging to itself
or another species then make sure "move()" is implemented in the
appropriate place. Make sure any calls to actions inside species
blocks or references to species have implementations. Make sure each
functionality declared in classes in {key-mermaidClassDiagramScript}
has been implemented in the appropriate species. REFLECT and IMPROVE
the script based on your reflection. Find and remove any GAML errors.
Memorise this as {key-gamlScriptDraft2}.
```

I proceeded to introduce a third step in GAML code generation, to ensure the generated code meets the objectives defined at the start. This step commands the model to facilitate for users to change experimental factors and see output. A check is also conducted to ensure the code encompasses all UML actors. Another advantage of this prompt is that it provides another opportunity to fix half-implemented sections of the GAML script, this may happen if the script is particularly verbose. Again, there are more code snippets/examples to reduce ambiguity. I found this approach to work very well and would like to draw attention to the progression of GAML code generation across the three implementation prompts in *mistral-nemo_flu_test_3_20241228-032321.json* -- the third prompt contains an attempt to produce output (infection curves) that align with the respective key-output. The third implementation prompt can be found below:

```
Build upon {key-gamlScriptDraft2}. Each "species" (and the "global"
block) must include all required attributes and logic to compute and
display {key-outputs} within the "experiment" block. Additionally, the
code must provide the necessary attributes and logic to allow users to
manipulate all {key-experimental factors}. Make sure all memorised
{key-umlActors} and their corresponding UML User Stories have been
implemented. Make sure the code meets the memorised {key-objectives}
and measures statistics for the memorised {key-objectives} AND
memorised {key-hypotheses}. All of this logic must be correctly
implemented in the appropriate code blocks. Any species you want to
visually display must be declared in the "display" block in
"experiment" and must have a corresponding "aspect" in their species
block, in the format: `aspect base {\n\t\tdraw circle(size) color:
```

```
color;\n\t}`. If {key-outputs} would benefit from txt/csv files, add
logic to output them. Finally, make sure all species have been
"created" inside the "init" block of the "global" block, using the
following syntax "create <species_name_1> number:
<initial_population_of_species_name_1>;". Memorise this as {key-
gamlScript}.
```

Note: between testing the second and third implementation prompts, I came across issues of the code not repeating GAML keywords and functions it had used prior. This was an issue as some tokens must be used repeatedly to get valid GAML, for example if you have multiple "species" the species keyword must be used many times, likewise if there is a commonly used helper function it may also be invoked many times. To solve this issue, I reduced the repeat penalty from 1.1 (Ollama default) to 1.05. As mentioned earlier, I have noted my final configuration in the appendix.

I have tried to consolidate any markdown output where possible – to reduce the output. I have also removed many confirmation messages to make the script succinct, the ones remaining are because the models have sometimes confused the corresponding keys.

After the refinement process; the number of tokens to go through an average EABSS script was around ~23-25k tokens, this is due to going through multiple iterations of GAML code generation to achieve more detailed code with better coverage and less syntax errors. I think that until smaller models capable of "reasoning" with better chain-of-thought (e.g. a smaller ChatGPT o1) are released, this approach of iteratively developing with the model is probably the best thing to do.

Additional Insights from using the Refined Script with Mistral-NeMo and Other Models

I observed some intriguing behaviour with Mistral-NeMo. Specifically, the model occasionally employs weighted sums as a criterion to decide between various potential outcomes for defining a given parameter. The LLM can accurately calculate these weighted sums—although not with perfect consistency—and then use them as a comparator to determine which outcome to proceed with. Please find an example below, taken on 22nd December 2024:

```
**Criteria for Ranking Potential ABSS Experimental Factors:**

1. **Feasibility**:
   - *Definition*: The practicality of implementing and testing the experimental factor within the study's scope
and resources.
   - *Scale*: 1-5 (1 being highly impractical, 5 being very feasible)
   - *Weight*: 0.3

2. **Novelty**:
   - *Definition*: The extent to which the experimental factor pushes the boundaries of current knowledge in ABSS
 or cellular automata modeling.
   - *Scale*: 1-5 (1 being low novelty, 5 being high novelty)
   - *Weight*: 0.3

3. **Practical Applicability**:
   - *Definition*: How well the experimental factor addresses real-world problems or provides insights relevant t
o stakeholders.
   - *Scale*: 1-5 (1 being low practical applicability, 5 being high practical applicability)
   - *Weight*: 0.4

*Ranking Score*: Each experimental factor will be scored based on the average of its weighted criteria scores.

*Example Calculation*:

| Experimental Factor | Feasibility (0.3) | Novelty (0.3) | Practical Applicability (0.4) | Ranking Score |
|---|---|---|---|---|
| IDLC | 4.5 | 3.8 | 4.2 | **4.17** |
| PRC | 4.0 | 4.5 | 3.8 | **4.07** |
| BRV | 4.2 | 4.0 | 4.5 | **4.23** |
| ICP | 4.5 | 3.5 | 4.0 | **4.07** |
| NS | 4.0 | 4.8 | 4.2 | **4.33** |
| SD | 4.8 | 3.2 | 4.5 | **4.29** |

Based on this ranking, the top three experimental factors to carry forward would be Neighborhood Size (NS), Birth
 Rule Variance (BRV), and Simulation Duration (SD).
```

I decided to re-install the Llama3.1 8B model (Ollama default version with Q4_K_M quantisation) (https://ollama.com/library/llama3.1), after coming across a Git issue relating to poor Llama3.1 performance in Ollama/llama.cpp a few weeks ago. Unfortunately, I cannot seem to locate the Git issue now, and if I recall correctly, it was a discussion to make a fix to a configuration file.

After running Llama3.1 and Mistral NeMo on the refined prompts I noticed some similarities and differences in behaviour.

Llama3.1 struggles to provide interesting criteria in the study outline phase of analysis and is not as good at identifying task-specific key-output, and often offers more general answers, an example can be found in *llama_conway_test_3_20241223-184421.json*. However, I have noticed that it is generally better at generating syntactically correct mermaid.js code with the new, refined UML generation prompts. This is evident with the syntactic quality of responses in all the attached llama3.1 test evidence.

Both models struggle with implementing the finer details of the models. However, Mistral NeMo seems to have a better grasp of how to implement code that allows the user to view key outputs, manipulate the experimental factors and test the hypotheses outlined prior. It also appears that Llama3.1 is worse than Mistral NeMo when it comes to implementing interactions from UML sequence diagrams into GAML and translating UML actors into GAML. NeMo is better at inferring extra classes that are necessary in a class diagram, an example being the responses in *predator_prey_test_5_20241227-231107.txt* (please find the class diagram in one of the figures above). This may be due to the many factors; such as a difference in the quantity (and quality) of training data pertaining to key programming principles such as abstraction and decomposition.

Results

Overall, the results show progress in generating much more detailed code that is closer to meeting the original objectives and requirements.

With a suitably long context length, I have found key authenticity (the ability to reliably recall and reuse the values of memorised keys) to not be an issue.

Test runs from the 24th,27th and 28th December 2024 show good examples of a strong GAML scaffold, however, all models seem to struggle to implement the relationships between species (classes/actors). This stops the model from writing code that can work towards a bigger, common goal. Nevertheless, Mistral NeMo has shown the potential to solve this problem, by demonstrating that it can decompose various UML use cases and identify further actors/objects, which it then tries to implement in GAML code.

There is room for improvement, syntax errors persist, and it is not possible to list all syntax errors in the prompts. Next steps should be explored to try and address this, I consulted ChatGPT regarding the training data requirements for fine-tuning. It estimated that between 5,000 to 10,000 high-quality examples would be needed for a model the size of Mistral NeMo. The data should contain a variety of prompts that identify and correct syntax mistakes across a diverse set of simulation scenarios. I believe there may be other possible solutions which can be explored; for example, the script could ask the LLM to integrate some of the implementation models it suggested earlier in the "Model Scope" stage.

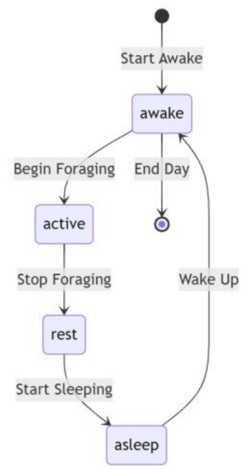
I have included a formatted copy of the generated GAML code from *mistral-nemo_bass_test_20241228-044436.json* in the appendix, as an example of the generated output.

Testing New LLMs

I tested StarCoder 15B in both its base version (https://ollama.com/library/starcoder:15b-base) and the Q6_K quantised version (https://ollama.com/library/starcoder:15b-q6_K), both tend to hallucinate and generate code even when presented with a straightforward prompt like: *"Work through each task list in the given order during the entire conversation. Answer with just 'yes' if you understand or 'no' if you don't understand."*

I conducted a quick test of the EABSS script on the Q6K quantised Mistral NeMo model (https://huggingface.co/QuantFactory/Mistral-Nemo-Base-2407-GGUF). The results were disappointing, with severe hallucinations, repetitions, and poor output (a small portion of which I saved as a text file on my desktop). Surprisingly, it performed worse than the default Mistral NeMo model from Ollama.

I also tested an EABSS scenario with the Mistral-Small model (https://ollama.com/library/mistral-small). The model performed well, UML and GAML generation were on a similar level to Mistral NeMo. The downside is that it has a lot more computational overhead and consumes more than 1.5x as much RAM (~20 GB) than Mistral NeMo.

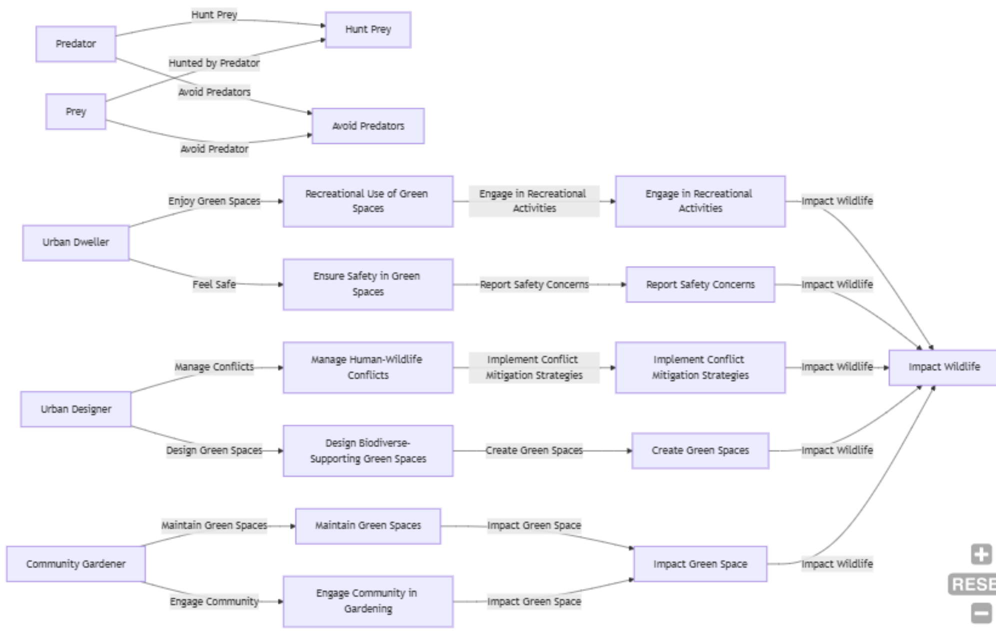


A state diagram generated by Mistral-Small

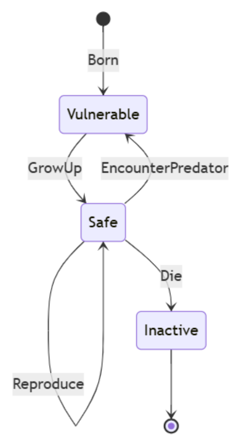
I finally managed to write a working notebook to try one of the existing GAML code generators on HuggingFace (https://huggingface.co/Phanh2532/GAMA-Code-Generator-1.2) in Google Colab. The notebook, *run_huggingface_from_colab.ipynb*, can be found in this folder. I generated a basic predator-prey model. The overall GAML code generation looks promising with max_new_tokens set to 2000 and a repetition_penalty of 1.15. However, the context window of the GAMA code generator is small, so it is not useful for all the other EABSS tasks. Still, the model shows that fine-tuned models have potential for code completion.

I attempted to run Llama3.3 70B multiple times on HuggingChat to execute the EABSS script, but it couldn't complete the process. Despite this, the model shows promise. A notable drawback is that it can't run locally and is perhaps subject to rate-limiting, which could result in delays between—or even during—test runs of the EABSS script. A copy of one of attempts can be found at https://hf.co/chat/r/_HW66a3?leafId=2f4f8281-5c7b-4ad0-b4db-ada42aa76188.

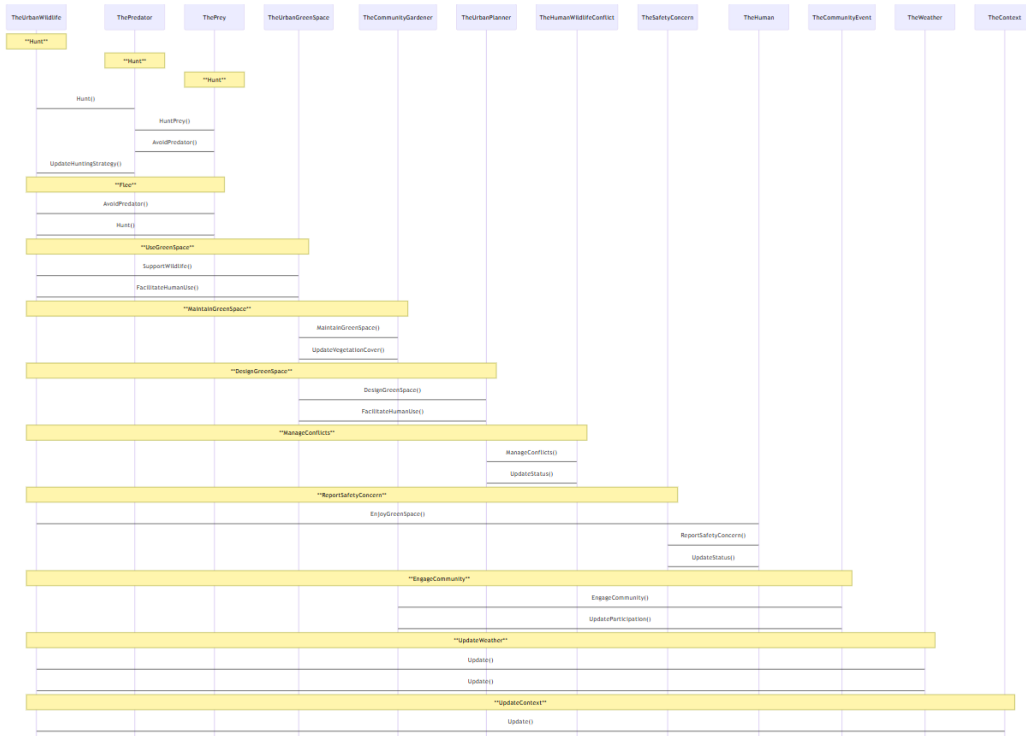
I also ran Mistral-Nemo-Instruct-2407 on HuggingChat (without allowing it to access the web) and I may do more investigation to see if there are any discrepancies, time permitting. An example conversation can be found at https://hf.co/chat/r/mBZjZPC?leafId=8a4538c6-7aa3-458b-9532-0c64c43ac9ed. I have included rendered diagrams from that conversation below:



Predator
Hunt Prey
Hunt Prey
Hunted by Predator
Prey
Avoid Predators
Avoid Predators
Avoid Predator
Urban Dweller
Enjoy Green Spaces
Recreational Use of Green Spaces
Engage in Recreational Activities
Engage in Recreational Activities
Impact Wildlife
Feel Safe
Ensure Safety in Green Spaces
Report Safety Concerns
Report Safety Concerns
Impact Wildlife
Urban Designer
Manage Conflicts
Manage Human-Wildlife Conflicts
Implement Conflict Mitigation Strategies
Implement Conflict Mitigation Strategies
Impact Wildlife
Impact Wildlife
Design Green Spaces
Design Biodiverse-Supporting Green Spaces
Create Green Spaces
Create Green Spaces
Impact Wildlife
Community Gardener
Maintain Green Spaces
Maintain Green Spaces
Impact Green Space
Impact Green Space
Impact Wildlife
Engage Community
Engage Community in Gardening
Impact Green Space



Born
Vulnerable
GrowUp
EncounterPredator
Safe
Die
Inactive
Reproduce



I researched whether it was possible for users to run their own models on HuggingChat, it is currently not possible to do so according to https://huggingface.co/spaces/huggingchat/chat-ui/discussions/517#66a24d940528bb6fd1d3a1fc.

Appendix

Using the Python EABSS Automation Script

1. The code can be found inside the *eabss_automation* folder. For convenience there is a copy of the refined EABSS script JSON file in this directory.
2. Create a Python virtual environment in the directory and activate the virtual environment. Then run: `pip install -r requirements.txt` to fetch dependencies.
3. To get the name of the model you would like to run, enter command: `ollama list`
4. To run the script enter (desired LLM's name can be found by running ollama: `python eabss_automation_bot.py --model <desired-llm-name e.g.mistral-nemo> --prompt-filepath <e.g. refined_prompts_as_json.json> --content-length <desired context length e.g. 32768> --temperature <desired temperature e.g. 0.4> --repeat-penalty <desired repeat penalty e.g. 1.05>`

5. You will be prompted to enter text that corresponds to key-topic, key-researchDesign, key-domain and key-specialisation.
6. The conversation will be printed to the terminal and saved as a JSON file in the same directory, with the following naming convention: `<desired-llm-name>_test_<timestamp>.json`

Recommended Model Parameters & Setting Parameters in Ollama

My final model parameter settings were:

- Temperature = 0.4
- Repeat penalty = 1.05
- Context length = 32768

There are two ways to set these values in Ollama. Firstly, you can make change values for just the current conversation by;

1. Starting your desired model by entering: `ollama run mistral-nemo`
2. Setting the parameter values:
    a. `/set parameter num_ctx <32768 or length of desired context window>`
    b. `/set parameter temperature <0.4 or desired temperature>`
    c. `/set parameter repeat_penalty <1.05 or desired repeat penalty>`
3. You can then verify your changes by following step (1) above and running: `/show parameters`

Or, alternatively you can edit the MODELFILE of the model (for persistent change) by;

1. Get the current configuration settings by running: `ollama show –modelfile <model_name>`
2. Copying the contents of the modelfile and pasting the contents into a new modelfile by following: https://github.com/ollama/ollama/blob/main/docs/modelfile.md#basic-modelfile
3. Adding these two lines below the line containing the <FROM> command: `PARAMETER num_ctx <32768 or length of desired context window>` `PARAMETER temperature <0.4 or desired temperature>`
4. You can then verify your changes by running `ollama show –modelfile <model_name>`

More information about MODELFILEs can be found at https://github.com/ollama/ollama/blob/main/docs/modelfile.md#build-from-a-gguf-file

Configuring KV Quantisation in Ollama

Installing flash-attention is a prerequisite to use KV caching in Ollama. After installing flash-attention, I proceeded to:

1. Open the ollama service file in Linux: `systemctl edit ollama.service`
2. Under service section of service file add:

   `OLLAMA_FLASH_ATTENTION=1`

   `OLLAMA_KV_CACHE_TYPE="q8_0" [other supported quantisations are listed at https://github.com/ollama/ollama/blob/main/docs/faq.md#how-can-i-set-the-quantization-type-for-the-kv-cache]`

3. Save and exit the service file
4. Run: `systemctl daemon-reload`
5. Then run: `systemctl restart ollama`

More information can be found at https://smcleod.net/2024/12/bringing-k/v-context-quantisation-to-ollama/ and https://github.com/ollama/ollama/blob/main/docs/faq.md#how-can-i-set-the-quantization-type-for-the-kv-cache.

Copy of Bass-Diffusion Inspired Model from mistral-nemo_bass_test_20241228-044436.json

While there are issues with the code below, such as declaring what would normally be the `experiment` block contents (`parameters`, `display` and `output`) in the `global` block, there is evidence of good decomposition, use of helper "species", and implemented methods that make for a solid scaffold overall.

```
model SustainablePracticeAdoption
global {
    init {
        float probabilityOfAwarenessCampaign <- 0.5;
        int awarenessCampaignStrength <- 1;
        float policyIncentiveStrength <- 0.1;
        float socialNetworkDensity <- 0.3;
        float greenSpacesBiodiversity <- 100;
        int numAgentUsers <- 100;
        int numCommunityLeaderUsers <- 10;
        int numEnvironmentalActivistUsers <- 5;
        int numLocalGovRepUsers <- 5;
    }
    create AgentUser number: numAgentUsers;
    create CommunityLeaderUser number: numCommunityLeaderUsers;
    create EnvironmentalActivistUser number:
numEnvironmentalActivistUsers;
    create LocalGovRepUser number: numLocalGovRepUsers;

    parameter awarenessCampaignStrength var: AgentUser.awarenessLevel;
    parameter policyIncentiveStrength var: AgentUser.adoptionRate;
    parameter socialNetworkDensity var: SocialNetwork.density;
    parameter greenSpacesBiodiversity var: GreenSpaces.biodiversity;
    parameter numAgentUsers var: int;
    parameter numCommunityLeaderUsers var: int;
    parameter numEnvironmentalActivistUsers var: int;
    parameter numLocalGovRepUsers var: int;

    output {
        AgentUser.adoptionRate;
        CommunityLeaderUser.influenceStrength;
        EnvironmentalActivistUser.campaignEngagement;
```

```
        LocalGovRepUser.policyInfluence;
        UrbanInfrastructure.accessibility;
        GreenSpaces.biodiversity;
        SocialNetwork.density;
        SocialNetwork.interactionStrength;

        txt file: AgentUserAdoptionRates.txt;
        csv file: AgentUserAdoptionRates.csv;
    }

    display AgentUser {
        species AgentUser aspect: awarenessLevel {
            draw circle(size: awarenessLevel * 5) color:
awarenessLevel < 5 ? "green" : awarenessLevel > 7 ? "red" : "yellow";
        }
        species AgentUser aspect: adoptionRate {
            draw circle(size: adoptionRate * 10) color: adoptionRate <
0.5 ? "blue" : adoptionRate > 0.8 ? "orange" : "gray";
        }
        species AgentUser aspect: advocacyStrength {
            draw circle(size: advocacyStrength * 3) color:
advocacyStrength < 2 ? "purple" : advocacyStrength > 4 ? "pink" :
"indigo";
        }
    }

    txt file AgentUserAdoptionRates.txt {
        AgentUser.adoptionRate foreach line: "%s\n";
    }

    csv file AgentUserAdoptionRates.csv {
        AgentUser.adoptionRate foreach line: "%s,%.2f\n";
    }
}

species AgentUser {
    float awarenessLevel <- 0;
    float adoptionRate <- 0;
    float advocacyStrength <- 0;
```

```
    reflex updateAwareness(awarenessCampaign: AwarenessCampaign) {
        awarenessLevel += awarenessCampaign.effectiveness *
awarenessCampaignStrength;
    }

    action adoptSustainablePractice(policyIncentive: PolicyIncentive)
{
        if (policyIncentive.incentiveType == 'Financial') {
            adoptionRate += policyIncentive.incentiveStrength;
            if (adoptionRate > 1) { adoptionRate = 1; }
        } else if (policyIncentive.incentiveType == 'Educational') {
            awarenessLevel += policyIncentive.incentiveStrength;
            if (awarenessLevel > 10) { awarenessLevel = 10; }
        }
    }

    action advocateForSustainablePractice(socialNetwork:
SocialNetwork) {
        advocacyStrength += socialNetwork.interactionStrength *
awarenessLevel;
        socialNetwork.influenceStrength += advocacyStrength *
awarenessLevel;
    }

    action engageWithEnvironmentalActivist(environmentalActivistUser:
EnvironmentalActivistUser) {
        if (awarenessLevel > 5 &&
environmentalActivistUser.campaignEngagement > 3) {
            environmentalActivistUser.designAwarenessCampaign('Join
the Movement!');
        }
    }

    action engageWithCommunityLeader(communityLeaderUser:
CommunityLeaderUser) {
        if (adoptionRate > 0.5 &&
communityLeaderUser.influenceStrength > 3) {
```

```
            communityLeaderUser.designIntervention('Community
Champion');
        }
    }
}

species CommunityLeaderUser {
    float influenceStrength <- 0;

    reflex monitorAdoptionRates(agentUsers: AgentUser[]) {
        float avgAdoptionRate = agentUsers.sum(user =>
user.adoptionRate) / agentUsers.length;
        if (avgAdoptionRate > 0.5) { influenceStrength += 0.1; }
    }

    action designIntervention(awarenessCampaign: AwarenessCampaign) {
        Intervention intervention = new Intervention('Community Leader
Intervention', awarenessCampaign.effectiveness * influenceStrength);
        return intervention;
    }

    action evaluateInterventionEffectiveness(intervention:
Intervention, agentUsers: AgentUser[]) {
        float interventionImpact = agentUsers.sum(user =>
user.adoptionRate * intervention.effectiveness) / agentUsers.length;
        if (interventionImpact > 0.1) { influenceStrength += 0.1; }
    }
}

species EnvironmentalActivistUser {
    float campaignEngagement <- 0;

    reflex designAwarenessCampaign(awarenessMessage: string) {
        AwarenessCampaign campaign = new
AwarenessCampaign('Environmental Activist Campaign', awarenessMessage,
campaignEngagement * 0.5);
        return campaign;
    }
```

```
    action optimiseEngagement(campaign: AwarenessCampaign) {
        campaign.engagementStrength += campaignEngagement * 0.1;
        if (campaign.engagementStrength > 3) { campaignEngagement +=
1; }
    }

    action collaborateWithCommunityLeader(communityLeaderUser:
CommunityLeaderUser) {
        if (campaignEngagement > 2 &&
communityLeaderUser.influenceStrength > 3) {
            communityLeaderUser.designIntervention('Activist
Collaboration');
        }
    }
}

species LocalGovRepUser {
    float policyInfluence <- 0;

    reflex assessPolicyImpact(agentUsers: AgentUser[],
policyIncentive: PolicyIncentive) {
        float policyImpact = agentUsers.sum(user => user.adoptionRate
* policyIncentive.incentiveStrength) / agentUsers.length;
        if (policyImpact > 0.1) { policyInfluence += 0.1; }
    }

    action designPolicyIncentive(adoptionRate: float) {
        PolicyIncentive incentive = new PolicyIncentive('Government
Incentive', adoptionRate * policyInfluence);
        return incentive;
    }

    action
collaborateWithEnvironmentalActivist(environmentalActivistUser:
EnvironmentalActivistUser) {
        if (policyInfluence > 3 &&
environmentalActivistUser.campaignEngagement > 2) {
            environmentalActivistUser.designAwarenessCampaign('Policy
Incentive Awareness');
```

```
        }
    }
}

species UrbanInfrastructure {
    float accessibility <- 0;

    action updateAccessibility(adoptionRate: float, agentUser:
AgentUser) {
        if (agentUser.adoptionRate > 0.5 && agentUser.adoptionRate >
accessibility) {
            accessibility = agentUser.adoptionRate;
        }
    }

    action updateAccessibilityForAgent(agentUser: AgentUser) {
        updateAccessibility(agentUser.adoptionRate, agentUser);
    }
}

species GreenSpaces {
    float biodiversity <- 100;

    action updateBiodiversity(adoptionRate: float, agentUser:
AgentUser) {
        if (agentUser.adoptionRate > 0.5 && agentUser.adoptionRate >
biodiversity) {
            biodiversity = agentUser.adoptionRate;
        }
    }

    action updateBiodiversityForAgent(agentUser: AgentUser) {
        updateBiodiversity(agentUser.adoptionRate, agentUser);
    }
}

species SocialNetwork {
    float density <- 0.3;
    float interactionStrength <- 0;
```

```
    reflex updateDensity(adoptionRate: float, agentUser: AgentUser) {
        if (agentUser.adoptionRate > 0.5 && agentUser.adoptionRate >
density) {
            density = agentUser.adoptionRate;
        }
        interactionStrength += agentUser.adoptionRate * 0.05;
    }

    reflex updateInfluenceStrength(socialPressure: float, agentUser:
AgentUser) {
        if (socialPressure > 0 && agentUser.adoptionRate >
interactionStrength) {
            interactionStrength = agentUser.adoptionRate;
        }
        agentUser.advocacyStrength += socialPressure * 0.1;
    }
}
```

Additional Resources

- A good resource to experiment with varying different LLM parameters: https://artefact2.github.io/llm-sampling/index.xhtml