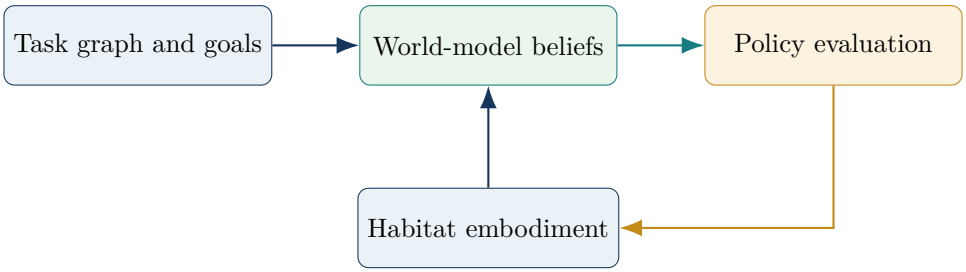
The Tea-Making Drone

A Technical Blueprint and User Guide for Hierarchical Active Inference in Habitat-Sim



Alexander D. Shaw

Computational Psychiatry & Neuropharmacological Systems (CPNS) Lab, Department of Psychology, University of Exeter

Version 1.0, July 2026

Lab Website: https://cpnslab.com

Project repository: https://github.com/alexandershaw4/active-inference-world-models

Related preprint: https://doi.org/10.5281/zenodo.21325216

Abstract

The tea-making drone is a simulated embodied agent that searches a realistic household environment for a mug, a kettle and a teabag, maintains explicit uncertainty about where these objects are located, infers hidden task states such as whether the kettle contains water and then executes a dependency-aware sequence of actions that fills, boils, pours, steeps and serves. The central contribution is a software architecture in which semantic beliefs, observation models, task dependencies, hidden-state inference and action selection remain distinct, inspectable and reusable, so that behaviour can change or adapt when evidence changes without rewriting the whole sequence by hand.

This document provides a detailed technical description of the current implementation, while also serving as a practical user guide for running, modifying and extending the software. The equations are presented alongside plain-language explanations and direct links to the corresponding Python modules, which makes the document suitable both as an online companion to the demonstration video and as a blueprint for researchers who want to reproduce the architecture in a different simulator or physical platform. Particular care is taken to state the scientific boundary of the present system: high-level search and task selection are belief-guided, the object and task-state models are probabilistic and the sensor evidence is obtained from Habitat semantic observations, while locomotion uses Habitat navmesh paths and manipulation remains abstract and kinematic rather than contact-rich.

Contents

1 Purpose and scope

The demonstration begins with a deceptively ordinary goal, which is to make and serve a cup of tea in an apartment. That goal becomes technically interesting once the agent is prevented from reading the correct answer directly from the simulator and is instead required to act through a world model. The mug, kettle and teabag are placed at hidden semantic sites, the agent begins with structured priors over plausible locations and every inspection changes its posterior beliefs through either positive evidence or negative evidence. Once the relevant objects have been found, the agent must still reason about states that are not visually obvious, including whether the kettle contains water, whether that water is hot, whether the mug contains hot water and whether the tea has brewed sufficiently to be served.

The software therefore addresses three linked problems. It must decide where to look when several object locations remain uncertain, it must decide which part of a long-horizon task should be addressed next and it must maintain a separation between the external state of the simulator and the internal beliefs used by the agent. These problems are handled by separate modules that communicate through deliberately narrow interfaces, which allows the active-inference logic to remain independent of Habitat and allows the task graph to remain independent of the implementation details of navigation, rendering and object placement.

This document describes the code contained in the Habitat port of the active-inference world-model project, with particular emphasis on `demos/run_habitat_make_tea.py` and the modules that it orchestrates. The implementation was inspected at the level of source code and validated using the included automated test suite, for which all 32 tests passed in the supplied repository snapshot. Habitat itself is an optional runtime dependency, so the simulator-independent inference and task components can be tested without loading a photorealistic scene.

Scientific boundary: the word “drone” describes the intended embodied agent, while the current Habitat video uses an invisible camera agent with a presentation marker

The video does not render a physical quadrotor body and it does not simulate propeller dynamics, aerodynamic control or contact-rich manipulation. Habitat provides a camera-bearing agent that follows navigable paths, while the third-person renderer adds a labelled marker so that viewers can see where the agent is. This choice keeps the scientific focus on world-model inference and long-horizon task organisation, while leaving a clear interface through which a future drone, mobile manipulator or robot arm can implement the same high-level actions.

1.1 What the demonstration shows

The complete episode shows a hierarchical loop in which the agent:

1. loads a scenario that defines semantic sites, object priors, hidden simulator truth and the physical Habitat scene;
2. builds independent location beliefs for the mug, kettle and teabag, rather than representing the environment through one monolithic state vector;
3. constructs a directed acyclic task graph that defines dependencies between search, manipulation and hidden-state transformations;

4. evaluates joint object-site inspection policies using a compact expected-free-energy-style objective that combines target probability, expected information gain, geodesic travel cost and top-down task relevance;
5. navigates to a selected vantage point, turns towards the site and uses semantic pixels plus spatial association to determine which unresolved objects are visible there;
6. updates all relevant object beliefs from the same observation, including negative-evidence updates for objects that were not found;
7. executes the next available task action, while automatically skipping an action when its probabilistic goal has already been met;
8. maintains Bernoulli beliefs over hidden states such as `kettle.contains_water` and updates these beliefs using noisy observations or probabilistic action effects;
9. records every decision, observation, posterior, task transition, object relation and rendering parameter in a synchronised JSON log;
10. produces a video that combines a third-person task view, an egocentric inset and overlays for current beliefs, policy scores, task progress and hidden tea states.

The system is best understood as a transparent research architecture rather than a benchmark-optimised end-to-end policy. Its value comes from the fact that each decision can be explained in terms of a belief, a task dependency or a scored candidate policy, which makes failure analysis and counterfactual testing substantially easier than they would be in a single opaque network.

2 Conceptual architecture

2.1 A generative world model rather than a scripted animation

A scripted tea-making animation can be produced by hard-coding a sequence such as "go to the counter, pick up the kettle, fill it, boil it and pour". That approach can look convincing when every object is where the script expects it to be, although it cannot explain what should happen when the mug has been moved, when the kettle is already full or when an inspection fails to reveal the expected object. The present architecture instead represents uncertainty explicitly and uses observations to determine how the sequence should unfold.

At the highest level, the agent maintains beliefs about hidden states, uses those beliefs to evaluate candidate actions and then samples the environment to obtain new evidence. The environment contains the true object locations and hidden tea states, although those values are not passed directly to the inference layer. The distinction can be written schematically as

$$\underbrace{s^\star}_{\text{environment truth}} \longrightarrow \underbrace{o}_{\text{sensor observation}} \longrightarrow \underbrace{q(s)}_{\text{agent belief}} \longrightarrow \underbrace{\pi}_{\text{selected policy}} \longrightarrow \underbrace{s^{\star\prime}}_{\text{changed environment}} . \tag{1}$$

Here, $s^\star$ denotes the external state of the simulator, o denotes an observation, $q(s)$ denotes the agent's posterior belief and π denotes an action or policy. The loop is active because action changes which observations become available and can also change the world itself, for example by moving the mug or filling the kettle.

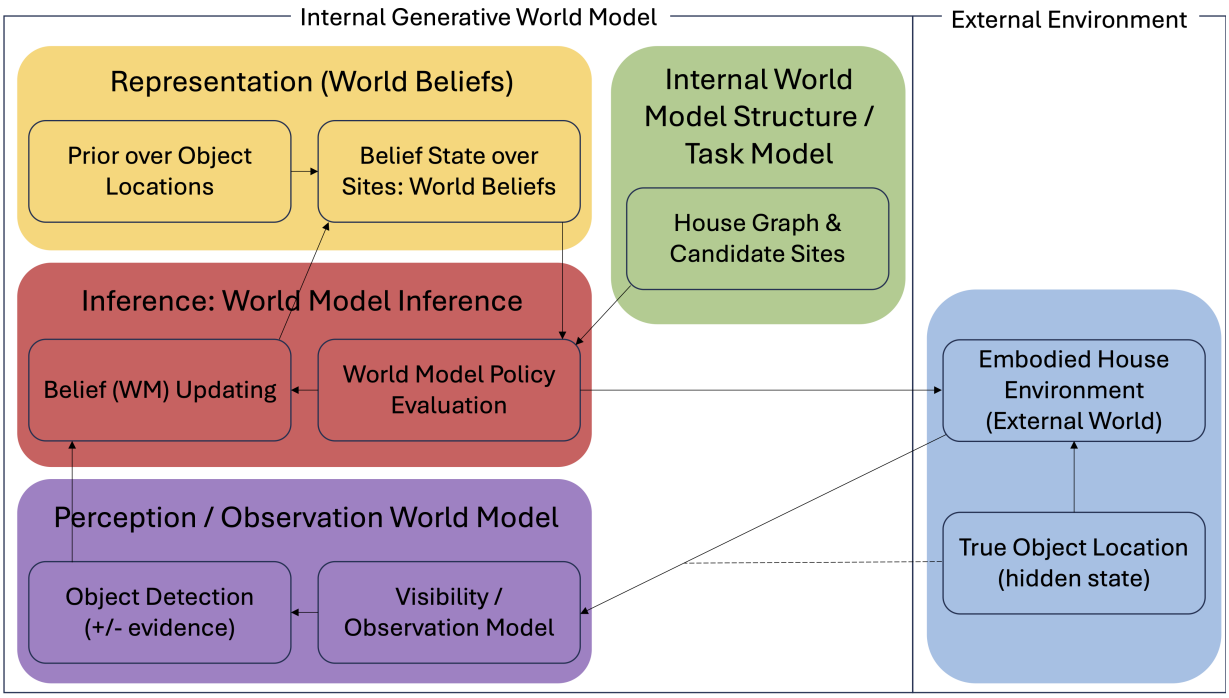


Figure 1: The general active-inference world-model architecture from which the household demonstration is developed. The current software extends the object-location layer with multiple independent objects, a task graph and probabilistic hidden task states.

2.2 Four software layers

The implementation is organised around four layers that should remain conceptually separate even when they are executed within one Python process.

Table 1: The principal layers of the tea-making architecture.

Layer	**Question answered**	**Principal modules**
Task structure	What must eventually be achieved, which actions depend on which prerequisites and which goal states allow actions to be skipped?	`tasks/actions.py`, `tasks/task_graph.py`, `tasks/household_tasks.py`
World model and inference	Where are the objects likely to be, what hidden properties are likely to hold and how should evidence update those beliefs?	`world_model/object_location_belief.py`, `world_model/latent_state_belief.py`, `world_model/object_state.py`, `world_model/object_registry.py`
Policy selection and execution	Which object-site inspection should be performed next, how does task relevance alter search and how are completed actions applied to beliefs?	`planning/semantic_efe.py`, `planning/semantic_policy.py`, `world_model/multi_object_search_agent.py`, `world_model/household_task_executor.py`
Embodiment and presentation	How are paths, cameras, semantic observations and abstract object actions realised, then rendered and logged?	`env/habitat_house_environment.py`, `presentation/multiview.py`, `presentation/overlay.py`, `demos/run_habitat_make_tea.py`

Design principle: the task layer says what should happen, while the environment adapter decides how it happens

A `TaskAction` contains a declarative verb such as `SEARCH`, `PICK`, `FILL` or `POUR`, together with an object, an optional target site and structured parameters. It does not contain Habitat calls, motor commands or rendering logic. This separation is what makes it possible to preserve the same task graph when Habitat is replaced by PyBullet or a physical robot, provided the new environment supplies implementations for the required action interface.

2.3 Runtime information flow

The full runner constructs the object registry and task graph before the episode begins, then repeatedly asks the task executor for the next action. When that action is a search, the runner creates a multi-object search agent over all unresolved objects on the task frontier and asks it to score every possible object-site pair. When the action is practical, such as picking up the kettle or pouring water, the runner delegates the physical or symbolic effect to Habitat and delegates the internal belief transition to the task executor.

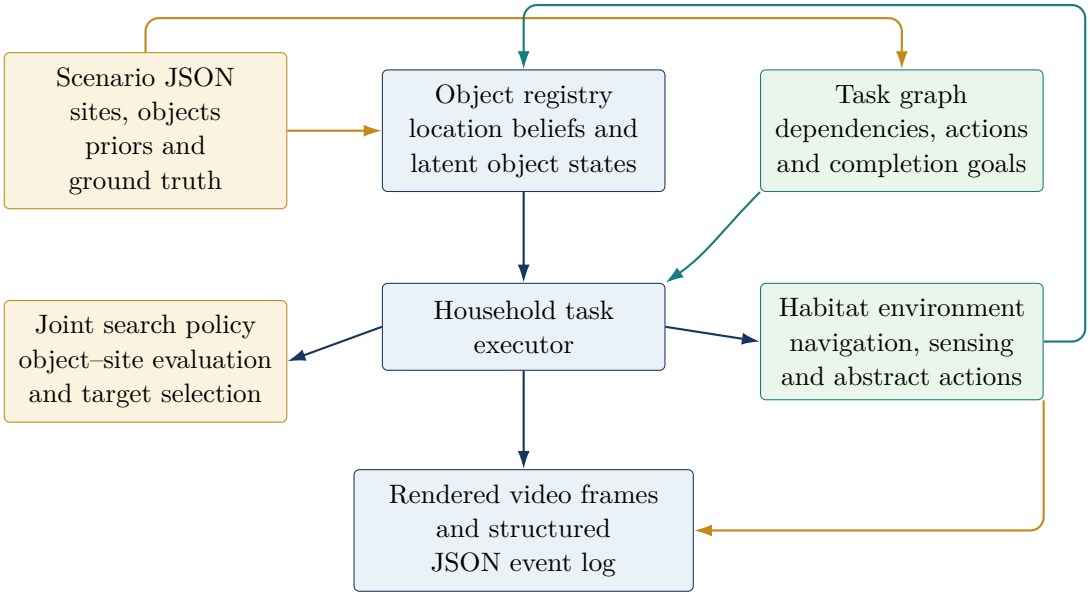


Figure 2: Runtime information flow during the complete tea-making episode. The household task executor coordinates the task graph, object beliefs and Habitat actions, while the output system records the resulting belief updates, task transitions and rendered frames. The arrows represent software-level data and control dependencies rather than layers within a single neural network.

3 Scenario representation and semantic world structure

3.1 Semantic sites

The agent does not search every continuous point in the apartment, but instead searches a smaller set of meaningful sites such as the kitchen counter, dining table, side table, living-room table and console table. Each site contains a surface position, from which objects are spawned or placed and a navigable vantage position, from which the agent can inspect the surface. This distinction is important because a visually meaningful object location may not itself be on the navigation mesh, while the corresponding viewpoint must be reachable by the agent.

Each site also contains category-specific affordance values, which are used as semantic priors over object location. A kitchen counter can therefore have a high prior for a kettle and a lower prior for a mug, while a console table can carry a relatively high prior for a teabag. These numbers do not describe physical affordances in the narrow manipulation sense. They are contextual probabilities or plausibility weights that encode where a given object category is expected to be found.

For object category c and semantic site i, the scenario provides a non-negative semantic weight a_{ic}. The initial location belief is constructed as

$$q_0(L_c = i) = \frac{\max(a_{ic}, \varepsilon)}{\sum_{k=1}^{K} \max(a_{kc}, \varepsilon)}, \tag{2}$$

where K is the number of candidate sites and ε is a small floor, set to 10^{-4} by default. The floor prevents any site from becoming impossible before evidence is observed, which is useful when a prior is incomplete or a scene has been rearranged.

3.2 Objects, affordances and hidden truth

Each object specification contains a stable name, a semantic category, a hidden true site, a unique semantic identifier and a ranked set of candidate asset names. It also contains a set of action affordances, such as `graspable`, `container`, `fillable`, `pourable` or `heatable`. These affordances are checked by the task executor before actions are completed, so an action that attempts to pour from a non-pourable object or insert into a non-container object fails explicitly rather than silently corrupting the state.

The scenario also separates state priors from hidden simulator truth. For example, the kettle can begin with

Listing 1: Illustrative hidden-state fields for the kettle.

```
"state_priors": {
  "contains_water": 0.35,
  "water_hot": 0.01
},
"hidden_state_truth": {
  "contains_water": false,
  "water_hot": false
}
```

The prior of 0.35 expresses the agent's uncertainty about whether the kettle contains water, while the hidden truth specifies that the simulated kettle is actually empty. The task agent receives the truth only indirectly through an observation produced by `observe_object_state`. This separation allows the same task graph to behave differently in the filled-kettle counterfactual, where positive evidence raises the posterior above the goal threshold and the fill action is skipped.

Design principle: scenario files contain world-specific facts, while inference modules contain reusable rules

Coordinates, scene identifiers, semantic IDs, asset candidates and hidden initial states belong in the scenario. Bayesian updates, task dependencies and expected-free-energy scoring belong in reusable Python modules. Keeping those roles separate prevents one photorealistic apartment from becoming hard-coded into the scientific architecture.

3.3 Schema version 2

The tea-making scenario uses schema version 2, which extends the earlier single-target format to support several objects with independent location beliefs. The loader in `utils/scenario_io.py` remains backwards compatible with schema version 1, although new household tasks should use the multi-object representation because a single inspection can then provide evidence about several unresolved objects.

A minimal schema-v2 scenario has the following shape:

Listing 2: Minimal structure of a schema-v2 scenario.

```
{
  "schema_version": 2,
  "name": "make_tea_replica_cad",
  "habitat": {
    "dataset_config": "PATH_TO_DATASET_CONFIG",
```

```
    "scene_id": "Baked_sc1_staging_00",
    "start_position": [1.27, 0.05, 4.19]
  },
  "objects": [
    {
      "name": "mug",
      "category": "mug",
      "true_site": "side_table",
      "semantic_id": 9001,
      "template_search": "frl_apartment_cup_01",
      "affordances": ["graspable", "container", "fillable"]
    }
  ],
  "sites": [
    {
      "name": "side_table",
      "room": "living_area",
      "position": [-2.23, 0.83, 3.67],
      "vantage_position": [-0.90, 0.05, 1.05],
      "affordances": {"mug": 0.15}
    }
  ]
}
```


The full supplied scenario adds ranked asset candidates, exclusions, position offsets, task relevance, state priors and hidden task-state truth. The asset resolver chooses the best available ReplicaCAD template at runtime and records the selected handle in the episode log, while a publication run can freeze those choices into the scenario for exact reproducibility.

4 Object-location beliefs and evidence accumulation

4.1 Independent categorical beliefs

For each tracked object o, the world model stores a categorical posterior over the K semantic sites,

$$\mathbf{q}^{(o)} = [q(L_o = 1), \ldots, q(L_o = K)], \qquad \sum_{i=1}^{K} q(L_o = i) = 1. \tag{3}$$

The implementation is provided by `ObjectLocationBelief`, while `ObjectState` binds that belief to object affordances, current symbolic location, containment relationships and hidden task-state beliefs. The use of independent object posteriors is deliberately simple and interpretable. It does not model correlations such as the possibility that the mug and teabag are likely to be stored together, although the registry can update all objects from one shared view and a future extension can replace the factorised representation with a structured joint model.

4.2 Positive evidence

When an object is detected at site j, the current implementation collapses the posterior close to a point mass,

$$q^{+}(L_o = i) \propto \begin{cases} 1, & i = j, \\ \varepsilon, & i \neq j, \end{cases} \tag{4}$$

where ε is the same small probability floor used during initialisation. The confirmed site is then stored in the object state and the corresponding search node in the task graph can be completed. This update is appropriate for the present simulator because semantic IDs provide highly reliable object identity once a sufficient number of pixels is visible and the object is confirmed to lie at the inspected site.

4.3 Negative evidence

Negative evidence is the more important feature of the search demonstration because an inspection that fails to reveal an object should still change what the agent does next. When site j is inspected and object o is not detected, the likelihood model assigns a small miss likelihood μ to the hypothesis that the object is nevertheless at that site,

$$P(o_j^{-} \mid L_o = i) = \begin{cases} \mu, & i = j, \\ 1, & i \neq j, \end{cases} \tag{5}$$

which produces the posterior

$$q^{-}(L_o = i) = \frac{q(L_o = i) P(o_j^{-} \mid L_o = i)}{\sum_{k=1}^{K} q(L_o = k) P(o_j^{-} \mid L_o = k)}. \tag{6}$$

The default value is $\mu = 0.05$, which means that a negative inspection makes the inspected site substantially less likely without making it impossible. A non-zero miss likelihood is valuable because the camera may have been poorly positioned, the object may have been occluded or the semantic pixel threshold may have been too strict. The posterior therefore shifts away from the inspected site while retaining a small possibility that the observation was a miss.

Implementation note: one inspection can update several unresolved objects

The environment returns a dictionary of detections for the unresolved objects that were considered during an inspection. The object registry applies a positive or negative update to each object for which the observation made an explicit claim. An omitted object means "no observation was available", whereas a value of `False` means "the site was inspected and the object was not detected". This distinction prevents missing data from being treated as negative evidence.

4.4 Entropy as uncertainty

The uncertainty of an object-location belief is measured by categorical entropy,

$$\mathcal{H}\left(\mathbf{q}^{(o)}\right) = -\sum_{i=1}^{K} q(L_o = i) \log q(L_o = i). \tag{7}$$

Entropy is high when probability mass is distributed across several sites and low when one site dominates. The policy evaluator uses this quantity to estimate the information that an inspection is expected to provide, while the video overlay visualises the same belief distribution as bars that change after each observation.

5 Expected-free-energy-style search

5.1 The candidate policy space

A search policy is defined at a semantic level rather than as a sequence of motor commands. For a single object, each candidate policy has the form

$$\pi_j = \text{inspect semantic site } j. \tag{8}$$

For several unresolved objects, the policy space becomes a Cartesian product,

$$\Pi = \{\pi_{o,j} : \text{inspect site } j \text{ for object } o\}. \tag{9}$$

The selected object primarily determines which belief and task-relevance term are used for scoring. Once the site is inspected, the observation can update every unresolved object that is visible or absent in that view, so the execution of a policy can be more informative than its label suggests.

5.2 Expected entropy after an inspection

For a candidate site j, the implementation approximates two possible outcomes. If the target is present and detected, the posterior entropy is treated as zero. If the target is absent, the current belief is updated using the miss-likelihood rule in equation (6). The expected posterior entropy is therefore

$$\mathbb{E}[\mathcal{H}_{\text{after}} \mid \pi_j] \approx q(L_o = j) \cdot 0 + (1 - q(L_o = j))\, \mathcal{H}\left(\mathbf{q}^{(o,-j)}\right), \tag{10}$$

where $\mathbf{q}^{(o,-j)}$ is the posterior that would follow negative evidence at site j. The anticipated information gain is then

$$I_{o,j} = \mathcal{H}\left(\mathbf{q}^{(o)}\right) - \mathbb{E}[\mathcal{H}_{\text{after}} \mid \pi_{o,j}]. \tag{11}$$

This quantity rewards inspections that are expected to reduce uncertainty, even when the object is not found. A site with moderate target probability can therefore be attractive when ruling it out would concentrate belief strongly elsewhere.

5.3 Pragmatic value, epistemic value and travel cost

The base policy score is implemented in `planning/semantic_efe.py` as

$$\mathcal{G}_{o,j}^{\text{base}} = w_d d_j - w_p q(L_o = j) - w_e I_{o,j} + w_a \, \mathbb{I}[j \text{ already inspected}], \tag{12}$$

where lower values are preferred. The terms have direct interpretations:

- d_j is the estimated travel distance from the current agent position to the site's vantage position, using Habitat geodesic distance when available and Euclidean distance as a fallback;
- $q(L_o = j)$ is the pragmatic probability that the target object is at the site;
- $I_{o,j}$ is the expected information gain from inspecting the site;
- the final indicator penalises repeated inspection of a site that has already produced negative evidence.

The default weights in the current implementation are

$$w_d = 0.20, \qquad w_p = 3.00, \qquad w_e = 1.50, \qquad w_a = 1.00. \tag{13}$$

The expression is described as expected-free-energy-style because it captures the central pragmatic and epistemic trade-off in a compact engineering objective, while it does not instantiate the full discrete-state active-inference generative model and message-passing scheme used in canonical formulations. This distinction is deliberate and should be retained in public descriptions.

5.4 Top-down relevance from the task graph

The task graph influences search by measuring how much unfinished work depends on locating each object. For the search action associated with object o, the raw relevance is

$$\tilde{r}_o = 1 + N_{\text{descendants}}^{\text{unfinished}}(o), \tag{14}$$

and the normalised relevance is

$$r_o = \frac{\tilde{r}_o}{\max_{o'} \tilde{r}_{o'}}. \tag{15}$$

The joint policy score becomes

$$\mathcal{G}_{o,j} = \mathcal{G}_{o,j}^{\text{base}} - \lambda r_o, \tag{16}$$

where $\lambda = 0.75$ by default in the Habitat tea runner. An object that unlocks more downstream actions receives a larger bonus, although a distant or highly improbable site can still lose to a more informative alternative. This arrangement connects the lower-level search policy to the structure of the long-horizon task without forcing the task graph to specify a fixed search order.

The selected decision is simply

$$(o^\star, j^\star) = \arg\min_{o,j} \mathcal{G}_{o,j}. \tag{17}$$

All candidate scores are retained in the decision object and written to the episode log, which means that a viewer can inspect not only the chosen policy but also the alternatives that were rejected.

Design principle: hierarchy should bias search without destroying epistemic flexibility

The task graph contributes a relevance bonus rather than dictating a fixed target. This means that an urgent object is more likely to be searched for, although the agent can still choose another object-site pair when it offers substantially better information or requires much less travel. The same principle can be extended to deadlines, energy budgets or user preferences without replacing the underlying location-belief model.

6 The household task graph

6.1 Actions as declarative nodes

The task layer defines a small simulator-independent vocabulary that includes `SEARCH`, `INSPECT`, `PICK`, `NAVIGATE`, `PLACE`, `INSERT`, `FILL`, `POUR`, `USE` and `WAIT`. Each action is stored in a `TaskNode` with a tuple of dependencies and a status that can be pending, active, complete or failed. The resulting `TaskGraph` is validated for missing references and cycles when it is created.

A pending action a_i is ready when all of its dependencies are complete,

$$\text{ready}(a_i) \iff \forall a_k \in \text{dep}(a_i), \quad \text{status}(a_k) = \text{complete}. \tag{18}$$

Insertion order is retained, which gives deterministic behaviour when several actions are equally ready, while parallel search branches remain available to the joint search planner. The graph can therefore express both a simple linear routine and a larger hierarchy in which several prerequisites are discovered or prepared independently before they converge on a later action.

6.2 The make-tea graph

The current graph contains 18 actions. It begins with three independent search branches, then coordinates object placement, hidden-state inspection and practical state transitions before serving the completed drink.

Table 2: Actions and dependencies in `build_make_tea_task()`.

Action ID	Kind	Human-readable role	Dependencies
`search_mug`	Search	Locate the mug	None
`pick_mug`	Pick	Pick up the mug	`search_mug`
`carry_mug`	Navigate	Carry the mug to the preparation site	`pick_mug`
`place_mug`	Place	Place the mug on the preparation surface	`carry_mug`
`search_teabag`	Search	Locate the teabag	None

Action ID	Kind	Human-readable role	Dependencies
`pick_teabag`	Pick	Pick up the teabag	`search_teabag`, `place_mug`
`carry_teabag`	Navigate	Carry the teabag to the preparation site	`pick_teabag`
`insert_teabag`	Insert	Put the teabag inside the mug	`carry_teabag`, `place_mug`
`search_kettle`	Search	Locate the kettle	None
`inspect_kettle_water`	Inspect	Observe whether the kettle contains water	`search_kettle`
`fill_kettle`	Fill	Fill the kettle unless the water goal is already satisfied	`inspect_kettle_water`
`boil_kettle`	Use	Boil the kettle after water is sufficiently probable	`fill_kettle`
`pick_kettle`	Pick	Pick up the hot kettle	`boil_kettle`, `insert_teabag`
`carry_kettle`	Navigate	Carry the kettle to the preparation site	`pick_kettle`
`pour_hot_water`	Pour	Pour hot water from the kettle into the mug	`carry_kettle`, `insert_teabag`
`place_kettle`	Place	Return the kettle to the preparation surface	`pour_hot_water`
`steep_tea`	Wait	Allow the tea to brew	`pour_hot_water`
`serve_tea`	Use	Mark the brewed tea as served	`steep_tea`, `place_kettle`

The graph contains several meaningful convergences. The teabag cannot be picked until the mug has been placed because insertion requires a known container at the preparation site. The hot kettle cannot be picked until the kettle has boiled and the teabag has been inserted, which prevents the task from carrying hot water to an incomplete tea station. The final serving action waits for both steeping and kettle placement, which ensures that the demonstration ends in a coherent household state rather than as soon as the tea-brewed flag changes.

6.3 Belief goals and automatic skipping

An action can declare a `belief_goals` field. Before returning an action for execution, the task executor checks whether every goal is already satisfied. For a condition that requires hidden state z to exceed threshold θ, the rule is

$$\text{satisfied}(z, \theta) \iff q(z = \text{true}) \geq \theta. \tag{19}$$

When the condition is satisfied, the action is completed with a structured `skipped` result and the

graph moves forward. The filled-kettle counterfactual uses this mechanism: a positive inspection moves the posterior for `contains_water` above 0.90, so `fill_kettle` is skipped and the same graph proceeds directly to boiling.

This mechanism is more general than a special-case `if kettle full` branch. Any practical action can be associated with a probabilistic goal, which means that future tasks can skip charging a battery, opening a door or fetching a tool whenever the corresponding belief is already sufficiently strong.

6.4 Belief preconditions

Actions can also declare probabilistic preconditions that must be satisfied before completion. Boiling requires

$$q(\texttt{kettle.contains\_water} = \text{true}) \geq 0.85, \tag{20}$$

while pouring requires both sufficient belief that the kettle contains water and sufficient belief that its water is hot. The task executor raises an explicit runtime error when a precondition is violated, which prevents the graph from claiming success after an incoherent transition.

7 Probabilistic hidden tea states

7.1 Bernoulli state beliefs

Location is not the only hidden variable that matters during tea making. Each task-relevant property is represented as a Bernoulli belief,

$$q(z) = \text{Bernoulli}(p), \qquad p = q(z = \text{true}), \tag{21}$$

with entropy

$$\mathcal{H}(p) = -p \log p - (1 - p) \log(1 - p). \tag{22}$$

The current state vocabulary includes:

- `kettle.contains_water` and `kettle.water_hot`;
- `mug.contains_water`, `mug.water_hot`, `mug.tea_brewed` and `mug.served`;
- `teabag.steeped`.

Each object can acquire additional latent states dynamically through `ensure_latent_state`, so the architecture is not limited to tea. A cupboard can have `open`, a battery can have `charged` and a plant can have `soil_moist`, provided the task graph and environment define the relevant observations and transitions.

7.2 Bayesian update from a noisy binary observation

Suppose a hidden state has prior probability $p = q(z = 1)$, while the sensor has sensitivity $\alpha = P(o = 1 \mid z = 1)$ and specificity $\beta = P(o = 0 \mid z = 0)$. For a positive observation, Bayes' rule gives

$$q(z = 1 \mid o = 1) = \frac{\alpha p}{\alpha p + (1 - \beta)(1 - p)}. \tag{23}$$

For a negative observation,

$$q(z = 1 \mid o = 0) = \frac{(1 - \alpha)p}{(1 - \alpha)p + \beta(1 - p)}. \tag{24}$$

The kettle inspection uses $\alpha = \beta = 0.95$ by default. These values are stored with the action, which allows a future implementation to assign different reliabilities to vision, weight sensing, temperature sensing or direct user reports.

7.3 Probabilistic action effects

Practical actions alter beliefs through an explicit success probability ρ. In the current implementation, an action intended to set a state to true predicts

$$q'(z = 1) = \rho, \tag{25}$$

while an action intended to set the state to false predicts

$$q'(z = 1) = 1 - \rho. \tag{26}$$

This model is intentionally direct: it treats the declared action reliability as the posterior confidence after the action rather than combining it with the previous belief through a full transition matrix. Filling the kettle therefore moves `contains_water` to 0.995, while pouring moves that belief close to false and moves the mug's `contains_water` belief close to true. The approach is easy to inspect and sufficient for the current demonstration, although a richer version could use state-dependent transitions,

$$q'(z') = \sum_{z} P(z' \mid z, a) q(z), \tag{27}$$

which would preserve uncertainty about both the previous state and the effect of the action.

7.4 Belief and simulator truth remain distinct

The Habitat adapter stores hidden truth separately from the object registry. When the agent inspects the kettle, the environment produces a noisy observation from that truth and the registry updates its posterior. When a practical action such as filling or boiling is executed, the task executor updates the agent's belief through the declared probabilistic effect and the environment

adapter separately updates simulator truth through `apply_state_truth_effects`.

Design principle: never let the planner read simulator truth merely because the simulator makes it convenient

A simulation often exposes exact state through an API, although passing that value directly to the planner removes the partially observed problem that the world model is intended to solve. The environment may use truth to generate an observation or to implement a state transition, while the task agent should reason from its own posterior. This rule is essential when moving from simulation to physical sensing because exact truth will no longer be available.

8 Perception and semantic evidence

8.1 Synchronous sensor suite

The Habitat environment creates an egocentric RGB camera, depth camera and semantic camera at the same sensor height. When the multi-view renderer is enabled, a second RGB sensor is attached behind and above the agent to provide the chase-camera view. The first-person RGB, depth and semantic observations remain the evidence available to the agent, while the third-person view exists only for presentation.

Each task-relevant object is assigned a unique semantic ID. During inspection, the environment counts the number of semantic pixels associated with each unresolved object,

$$n_o = \sum_{u,v} \mathbb{I}[S(u,v) = \mathrm{id}(o)]\,, \tag{28}$$

where $S(u,v)$ is the semantic image at pixel (u,v). A detection requires

$$n_o \geq n_{\min} \quad \text{and} \quad \mathrm{object\_at\_site}(o,j) = \mathrm{true}, \tag{29}$$

with $n_{\min} = 25$ pixels by default. The spatial association check prevents an object on a neighbouring surface from being assigned to the inspected site merely because it appears at the edge of the camera view.

8.2 Inspection as an active sensing movement

Navigation ends at a site's vantage position rather than at the surface itself. The environment then computes the yaw that faces the surface and rotates towards it in small increments until the angular error falls below the inspection tolerance. Only then is semantic evidence read. This separation makes the episode visually continuous and ensures that the observation corresponds to an intentional active-sensing pose rather than an arbitrary frame captured while the agent is still turning.

8.3 Current observation assumptions

The semantic observation model is reliable by construction because object identity is provided by simulator semantic IDs. It still includes three useful sources of uncertainty: the object must occupy enough pixels, the viewpoint can fail to reveal an occluded object and a negative inspection retains a non-zero miss likelihood. What it does not yet model is learned visual recognition, category confusion, false semantic labels or uncertainty over scene segmentation.

A natural next step is to replace the semantic-ID test with a probabilistic detector that returns likelihoods or calibrated confidence scores. The location-belief update can then use

$$q'(L_o = i) \propto q(L_o = i)P(o_t \mid L_o = i, \nu_t), \tag{30}$$

where ν_t represents the current viewpoint, visibility and detector state. The rest of the task architecture need not change.

9 Navigation and embodied control

9.1 What the current tea video uses

The tea-making runner uses Habitat's navigation mesh to obtain a shortest path from the current agent position to the selected site's vantage position. Candidate travel cost is therefore based on geodesic distance rather than straight-line distance whenever a valid path can be found. The resulting corner points are retained and the environment advances through them over many rendered frames.

At each frame, the agent computes the desired yaw towards the next path point. If the angular error is larger than the configured turn-before-move threshold, the agent rotates in place. Once it is facing sufficiently close to the route direction, it moves by at most the configured movement step. Translation and yaw are therefore smoothed separately, which avoids the visual discontinuities that arise when a camera snaps directly between navmesh points.

The default parameters used by the runner are listed in table 3. The command line deliberately exposes these values because photorealistic scene geometry and camera conventions can differ between Habitat builds.

Table 3: Selected runtime defaults in `run_habitat_make_tea.py`.

Parameter	Default	Role
`fps`	30	Output video frame rate
`movement_step`	0.018	Maximum translation per rendered frame
`turn_step`	2.5 degrees	Maximum yaw change per rendered frame
`max_frames`	50,000	Safety limit for a complete episode
`min_visible_pixels`	25	Semantic evidence threshold
`relevance_weight`	0.75	Strength of top-down task relevance in joint search
`view`	`multi`	Third-person view with first-person inset
`third_person_distance`	3.2 m	Chase-camera distance behind the agent
`third_person_height`	2.2 m	Chase-camera height
`third_person_pitch`	−22 degrees	Chase-camera pitch
`third_person_hfov`	78 degrees	Chase-camera horizontal field of view

10 Abstract manipulation and object relations

10.1 Kinematic action primitives

Habitat manipulation is implemented through explicit object-state changes rather than grasp synthesis and fluid dynamics. The environment can pick a visible object, attach it at a fixed offset in front of the agent, place it on a semantic surface, place it relative to another object or insert it into a container. The world model mirrors these changes symbolically through `held`, `current_site` and `contained_in` fields.

The separation between visual placement and symbolic relation is important. When the teabag is inserted into the mug, the environment positions the teabag relative to the mug so that the video communicates the action, while `ObjectState.mark_inserted` records that the teabag is contained in the mug and shares its current site. A later physics backend can replace the placement rule without changing the task semantics.

10.2 Tea-state transformations

Filling, boiling, pouring and steeping are currently symbolic or kinematic transformations because Habitat does not model water volume, thermal transfer or infusion chemistry. Their scientific role is to test whether a long-horizon task can depend on inferred hidden states and probabilistic transitions. The task requires the right preconditions, the agent belief changes with a declared reliability and the environment truth changes separately so that later observations can remain coherent.

The pour action illustrates this pattern particularly clearly. It requires a held, pourable kettle and a container destination, then it produces four belief effects: the kettle becomes unlikely to contain water, the kettle water becomes unlikely to remain hot, the mug becomes likely to contain water

and the mug water becomes likely to be hot. The action can therefore be analysed as a structured transition in a world model even though no fluid particles are simulated.

10.3 Affordance checking

Before a high-level action is completed, the executor verifies that the object has the required affordance. The checks include:

- `PICK` requires `graspable`;
- `INSERT` requires the destination to be a `container`;
- `FILL` requires `fillable`;
- `POUR` requires the source to be `pourable`, held and paired with a container destination;
- `USE` can declare an operation-specific affordance such as `heatable`.

These checks turn the object registry into more than a set of labels. It becomes a compact semantic model of what actions are possible, which is a necessary bridge between object identity and task planning.

11 The complete episode loop

The central runner in `demos/run_habitat_make_tea.py` is intentionally explicit. It does not hide task execution behind a large framework, which makes the control flow suitable for inspection and teaching. The logic can be summarised as follows:

Listing 3: Simplified pseudocode for the complete tea-making episode.

```
scenario = load_habitat_scenario(path)
registry = ObjectRegistry.from_specs(scenario.objects, scenario.sites)
graph = build_make_tea_task(preparation_site="dining_table")
executor = HouseholdTaskExecutor(graph, registry)
environment = HabitatHouseEnvironment(...)

observations = environment.reset()

while not executor.done:
    action = executor.next_action()

    if action.kind == SEARCH:
        relevance = executor.search_relevance()
        search = MultiObjectSearchAgent(
            sites=scenario.sites,
            registry=registry,
            required_objects=executor.pending_search_objects,
            task_relevance=relevance,
        )
        decision = search.choose_next(
            environment.get_agent_position(),
            environment.estimate_distance,
        )
```

```
        navigate_and_inspect(decision.site_name)
        detections = environment.inspect_objects(...)
        update = search.observe_site(decision.site_name, detections)
        confirm_any_resolved_searches(update)
        continue

    execute_environment_action(action)
    executor.complete_action(action)
    update_environment_truth_if_required(action)
    render_and_log_state()
```

Several details matter in the real implementation. Newly resolved objects are passed to `executor.confirm_search` while unresolved search nodes remain available for replanning. Practical actions are logged before and after completion, while the video holds selected frames for several output frames so that a viewer has time to read the overlays.

11.1 Search continues until the relevant object is found

The task executor may return `search_mug` as the current frontier action, although the multi-object search agent is allowed to evaluate every unresolved object whose search node remains incomplete. This arrangement means that one site inspection can resolve a different object first, after which its own search node is confirmed and the task graph can progress along that branch. The current frontier action remains incomplete until its object is found, which preserves task consistency while allowing information-efficient search.

11.2 The episode is event sourced

Every meaningful transition is written to a structured log. Search decisions record the selected object, selected site, total score, base score, task relevance, target probability, information gain, travel distance and every candidate alternative. Search observations record detections, visible pixel counts, posterior updates and newly resolved objects. Practical actions record parameters, graph state and object state before and after completion.

This event-sourced design makes it possible to reconstruct why the agent behaved as it did, compare alternative weighting schemes offline and create new visualisations without rerunning Habitat. It also establishes the right foundation for future evaluation, where metrics such as travel distance, number of inspections, entropy reduction, action skips and task completion time can be derived directly from logs.

12 Rendering and explanatory overlays

12.1 Multi-view presentation

The default output uses a full third-person chase-camera view with a synchronised first-person inset. The first-person image shows the observation available to the agent, while the third-person view makes navigation and object movement legible to a human viewer. Habitat agents have no

visible body, so a labelled 2D marker is placed at the expected agent position in the third-person image. This marker is not included in semantic sensing, collision checking or policy selection.

The user can select `-view first`, `-view third` or `-view multi`. Camera distance, height, pitch, field of view, inset scale and marker position are command-line parameters, while the final values are written into the JSON log to preserve reproducibility.

12.2 Belief and policy overlay

The search overlay displays the current categorical location belief for the selected object, the phase of the episode and the selected semantic site. When a decision is available, the panel can also display candidate scores and their components. The visual purpose is explanatory rather than decorative: the viewer should be able to see a belief move away from an inspected kitchen site after negative evidence and then collapse onto the side table when the mug is detected.

12.3 Task progress and hidden-state overlay

A second overlay displays task nodes and their current status, while a third displays the Bernoulli probabilities of hidden tea states. These panels make the transition from object search to state inference visible. In the empty-kettle scenario, the viewer can see `contains_water` fall after inspection, rise after filling, then change again after pouring. The video therefore communicates that the agent is maintaining an internal state model rather than merely moving objects along a fixed route.

13 Installation and execution

13.1 Repository and Python environment

The Habitat port is designed around Habitat-Sim 0.3.3 on Apple Silicon with Python 3.9. The repository keeps the Habitat environment separate from the original PyBullet dependencies because the available Habitat build constrains the Python version. From the repository root, the recommended environment setup is:

Listing 4: Creating the Habitat environment on Apple Silicon.

```
conda deactivate
CONDA_SUBDIR=osx-arm64 conda create -n habitat-aiwm \
  --override-channels \
  -c conda-forge \
  -c aihabitat \
  python=3.9 \
  "habitat-sim=0.3.3=py3.9_bullet_macos_*" \
  -y

conda activate habitat-aiwm
conda config --env --set subdir osx-arm64
python -m pip install -r requirements-habitat.txt
```

On macOS, the non-headless Habitat build should be used because the headless option relies on EGL. Other operating systems should use the Habitat build appropriate to their platform, while preserving compatibility with the repository's Python requirements.

13.2 Downloading ReplicaCAD

The photorealistic apartment and baked-lighting assets can be downloaded through Habitat's dataset utility:

Listing 5: Downloading ReplicaCAD data.

```
python -m habitat_sim.utils.datasets_download \
  --uids replica_cad_dataset replica_cad_baked_lighting \
  --data-path data
```

The baked-lighting scenes are useful for polished videos because they retain navigation meshes while treating most furniture as static. Before running the full task, the selected scene should be opened in Habitat's viewer and the scenario paths should be updated to match the local dataset installation.

13.3 Run simulator-independent checks first

The repository includes mock episodes that exercise inference and task logic without Habitat. These runs are useful when changing equations, priors or graph dependencies because they are fast and avoid scene-specific failure modes.

Listing 6: Fast simulator-independent checks.

```
python tools/preview_search_sequence.py
python demos/run_multi_object_search_mock.py
python demos/run_make_tea_mock.py
python demos/run_make_tea_mock.py \
  scenarios/make_tea_replica_cad_filled_kettle.json
```

The full automated test suite can be run with:

Listing 7: Running the test suite.

```
python -m pytest -q
```

13.4 Run the complete Habitat episode

Once the scenario paths and coordinates have been verified, the complete video is generated with:

Listing 8: Running the complete tea-making episode.

```
python demos/run_habitat_make_tea.py \
  scenarios/make_tea_replica_cad_working.json \
  --output outputs/habitat_make_tea_multiview.mp4
```

The runner writes two files with the same base name: an MP4 video and a JSON state log. A camera preview can be generated before committing to a complete episode:

Listing 9: Previewing the multi-view camera.

```
python tools/preview_multiview_camera.py \
  scenarios/make_tea_replica_cad_working.json
```

The preview is particularly useful after changing scene coordinates, chase-camera pitch or the marker anchor because a full tea episode can contain many thousands of frames.

14 Adapting the scenario

14.1 Choosing a scene and recording semantic sites

A new apartment requires one manual scene-design step. The user should choose meaningful surfaces, record their object positions and identify navigable vantage positions from which those surfaces are visible. Habitat uses the y axis as vertical, while each vantage point should lie on or very close to the navigation mesh.

A useful site should satisfy four conditions: the surface should be visually identifiable, the viewpoint should be reachable, the camera should see enough of the surface for semantic detection and the route should be visually coherent in the final video. The survey tools in `tools/habitat_scene_survey.py` and `tools/habitat_floor_survey.py` can assist with coordinate selection.

14.2 Changing object placement

The `true_site` field controls where an object is spawned, while the prior remains defined by the site's affordance values. Moving an object without changing the prior is a useful test of belief revision because the agent begins with its old contextual expectation and must recover through negative evidence. Changing both the true site and the priors creates a new environment model rather than a deliberate prior violation.

Each object must have a unique semantic ID. Position and vertical offsets can be used when an asset sits incorrectly on a surface, while template candidates and exclusions allow the runtime resolver to select an available kettle-like or teabag-like proxy without silently choosing a mug when a better asset exists.

14.3 Changing the preparation site

The task builder accepts a `preparation_site` argument and the runner exposes it through `-preparation-site`. The site must exist in the scenario. Object placement offsets may require adjustment when the surface size or orientation changes, although the logical graph does not need to be rewritten.

14.4 Changing priors and policy weights

Location priors are modified through site affordances. The task relevance weight is exposed on the command line, while the semantic EFE component weights are currently defaults inside `score_inspection_policy`. A more configurable public release could move these weights into the

scenario or a separate experiment configuration, which would make parameter sweeps easier to reproduce.

When tuning weights, the JSON candidate logs should be inspected rather than judging only the final video. A high pragmatic weight tends to send the agent to the most likely location, a high epistemic weight tends to favour inspections that sharply reduce uncertainty and a high distance weight tends to produce spatially efficient routes. The relevance weight controls how strongly unfinished downstream task structure biases which object is prioritised.

15 Validation and counterfactual tests

15.1 Unit and integration tests

The supplied repository includes tests for object-location inference, multi-object search, task-graph construction, task execution, hidden-state beliefs, scenario loading, asset resolution and multi-view rendering. The current snapshot passes all 32 tests, which provides a useful baseline before modifying the architecture.

Important test classes include:

- negative-evidence updates and object resolution;
- shared observations across multiple objects;
- dependency validation and cycle detection in task graphs;
- automatic action skipping when a belief goal is already satisfied;
- precondition enforcement for hidden-state actions;
- final object relations after tea-station preparation;
- different task trajectories for empty and filled kettle conditions;
- output geometry for first-person, third-person and multi-view frames.

15.2 The filled-kettle counterfactual

The most important behavioural validation is not simply that the empty-kettle episode completes. It is that the same task graph behaves differently when the hidden initial state changes. In the filled-kettle scenario, the positive inspection raises $q(\texttt{contains\_water} = 1)$ above the 0.90 fill goal. The executor marks `fill_kettle` as skipped and proceeds to boiling, which demonstrates conditional planning from inferred state rather than a fixed script.

15.3 Prior-violation tests

A second useful validation is to move an object to a low-prior site while retaining the original semantic priors. The agent should first inspect more plausible sites, incorporate negative evidence and redirect search as the posterior changes. This test directly exercises the claim that absence is treated as evidence rather than as a failed frame that leaves the policy unchanged.

15.4 Ablations

The architecture supports several informative ablations:

- set $w_e = 0$ to remove epistemic value and test purely probability-and-distance-driven search;
- set $\lambda = 0$ to remove top-down task relevance and compare search ordering;
- set the miss likelihood close to one to weaken negative evidence;
- replace geodesic distance with Euclidean distance to test the importance of navigable route geometry;
- remove action success uncertainty by setting all transition probabilities close to one;
- remove the semantic site-association check and quantify false assignment from peripheral visibility.

These ablations can be evaluated using total geodesic distance, inspections before resolution, entropy reduction per inspection, number of repeated sites, action skips and overall task completion.

16 Limitations and scientific boundaries

16.1 The observation model is idealised

Semantic IDs provide reliable object identity and avoid the difficult problem of learned visual detection. This is appropriate for isolating search and task inference, although it means that the current demonstration does not establish robustness to real camera noise, category confusion or domain shift. The next perceptual step should introduce probabilistic detection while preserving the same observation interface.

16.2 Object-location beliefs are factorised

Each object has an independent categorical posterior. This makes inference transparent, although it cannot express structured relations such as "the teabag is likely to be near the kettle" or exclusivity constraints such as "only one object can occupy this small site". A factor graph, scene graph or relational generative model could represent these dependencies while retaining semantic sites as interpretable variables.

16.3 Failure recovery is limited at the task level

The graph supports failed nodes, although the current tea runner primarily assumes that valid paths exist and actions complete when their checks pass. A mature household agent should represent retries, alternative methods, resource depletion and recovery plans. For example, a failed kettle fill could trigger reinspection, a missing teabag could trigger a substitute policy and an unreachable site could be removed from the candidate set.

16.4 The world model does not yet learn

Priors, observation reliability and action success probabilities are configured rather than learned from repeated episodes. The JSON logs provide the data needed to estimate these quantities, which suggests a natural empirical Bayes or hierarchical learning extension in which repeated experience updates semantic priors and transition models while preserving uncertainty.

17 Troubleshooting

Table 4: Common failure modes and likely remedies.

Symptom	**Likely cause**	**Suggested remedy**
Habitat-Sim cannot be imported	Wrong Python version, architecture or environment	Use the dedicated Python 3.9 Habitat environment and verify that the installed package matches the platform architecture
Scene or dataset config cannot be found	Scenario contains machine-specific absolute paths	Replace `dataset_config`, `object_config_dir` and fallback asset paths with local paths
No navigable path to a site	Vantage point is off the navmesh or isolated	Snap or re-record the vantage coordinate using the floor and scene survey tools
Agent arrives but detects nothing	Camera does not face the surface, object is occluded or pixel threshold is too high	Preview the inspection pose, adjust object placement or reduce `-min-visible-pixels` cautiously
Object appears in view but is assigned to the wrong site	Nearby surfaces are within the association tolerance	Reduce site tolerance, separate object positions or choose a more discriminating viewpoint
Kettle or teabag appears as a cup	No better local asset matched the candidate list	Run `tools/audit_tea_assets.py`, inspect available templates and freeze a preferred asset handle
Video jumps around corners	Movement or yaw step is too large	Reduce `-movement-step` and `-turn-step`, while retaining turn-before-move behaviour
Third-person camera looks too high or too low	Pitch convention or scene scale differs	Use `tools/preview_multiview_camera.py` and tune distance, height and pitch before a full run

Symptom	Likely cause	Suggested remedy
Task graph has no executable action	Dependency deadlock, failed node or unsatisfied state assumptions	Inspect the JSON graph status and verify dependencies, goal thresholds and preconditions
Fill action is unexpectedly skipped	Kettle posterior already exceeds the goal threshold	Inspect the state observation, prior and sensitivity or specificity values in the JSON log
Repeated site inspections occur	Negative evidence is too weak or inspected-site penalty is too small	Reduce the miss likelihood or increase the already-inspected penalty, then compare candidate logs

18 Summary

The tea-making drone provides a compact but extensible demonstration of how an embodied system can move beyond reactive control and fixed scripts by maintaining explicit beliefs about a structured world. Semantic priors guide the first search, negative evidence reshapes those priors, expected information gain competes with travel cost, task dependencies determine which discoveries matter and hidden-state inference allows the same graph to behave differently when the kettle is empty or already filled.

The current implementation is strongest when treated as a transparent blueprint. Its modules expose the precise boundary between environment truth, observation, posterior belief, policy selection and action effect. Habitat supplies a realistic visual environment and reliable semantic sensing, while the scientific architecture remains simulator-independent and can be tested without rendering. The manipulation layer remains abstract and navigation is currently navmesh driven, although both have clear replacement interfaces.

The broader design principle is that useful world models should not only predict what an agent will see. They should organise what the agent believes, determine what it needs to find out, represent what its actions are expected to change and preserve enough uncertainty to recover when the world violates its initial expectations. Making tea is a deliberately familiar task, although the same architecture can be applied to inspection, assistive robotics, laboratory automation, agriculture and any setting in which long-horizon behaviour depends on hidden state and active information gathering.

A Repository map

Listing 10: Principal files in the Habitat tea-making implementation.

```
active_inference_world_models_habitat_port/
|-- demos/
| |-- run_habitat_make_tea.py
| |-- run_make_tea_mock.py
```

```
| |-- run_habitat_multi_object_search.py
| `-- run_habitat_prepare_tea_station.py
|-- tasks/
| |-- actions.py
| |-- task_graph.py
| `-- household_tasks.py
|-- world_model/
| |-- object_location_belief.py
| |-- latent_state_belief.py
| |-- object_state.py
| |-- object_registry.py
| |-- multi_object_search_agent.py
| `-- household_task_executor.py
|-- planning/
| |-- semantic_efe.py
| `-- semantic_policy.py
|-- env/
| |-- habitat_house_environment.py
| |-- habitat_asset_resolver.py
| `-- habitat_objects.py
|-- presentation/
| |-- multiview.py
| `-- overlay.py
|-- scenarios/
| |-- make_tea_replica_cad_working.json
| `-- make_tea_replica_cad_filled_kettle.json
|-- tools/
| |-- preview_multiview_camera.py
| |-- audit_tea_assets.py
| |-- habitat_scene_survey.py
| `-- habitat_floor_survey.py
`-- tests/
```

B Notation

Table 5: Mathematical notation used in this document.

Symbol	Meaning
o	A tracked object such as the mug, kettle or teabag
j, i, k	Indices over candidate semantic sites
L_o	Hidden categorical location of object o
$q(L_o = i)$	Agent posterior probability that object o is at site i
z	A binary hidden task state such as `contains_water`
$q(z = 1)$	Posterior probability that hidden state z is true
$\mathcal{H}(\cdot)$	Shannon entropy of a categorical or Bernoulli belief
$I_{o,j}$	Expected information gain from inspecting site j for object o
d_j	Geodesic or fallback Euclidean travel distance to site j
r_o	Normalised task relevance of locating object o
$\mathcal{G}_{o,j}$	EFE-style score for joint object-site policy (o, j), where lower is preferred
μ	Miss likelihood used during negative-evidence location updates
α, β	Sensitivity and specificity of a binary state observation
ρ	Declared success probability of a practical state transition

C Configuration checklist for a publication run

Before generating a final public video, verify the following items:

1. the scenario uses portable or documented dataset paths rather than unexplained machine-specific paths;
2. every semantic site has a reachable vantage position and a camera angle that reveals the intended surface;
3. mug, kettle and teabag assets resolve to visually distinct templates or their proxy status is stated clearly;
4. object semantic IDs are unique and the visible-pixel threshold produces stable detections;
5. priors, miss likelihood, policy weights, state observation reliability and action success probabilities are recorded;
6. the multi-view camera has been previewed and the presentation-only agent marker aligns with the chase view;
7. the automated test suite passes after any source modification;
8. both the MP4 and JSON log are archived, with the exact scenario file and software commit used to generate them;
9. public text states that navigation is navmesh based and manipulation is abstract unless a later backend has replaced those components;

10. the final episode is checked for task completion, coherent object relations and plausible hidden-state transitions.

D Selected implementation parameters

Table 6: Defaults that materially influence inference or presentation.

Parameter	**Current value**	**Interpretation**
Location prior floor	10^{-4}	Keeps all semantic sites possible before decisive evidence
Negative-evidence miss likelihood	0.05	Probability of an absence observation when the object is actually at the inspected site
Distance weight w_d	0.20	Cost of travel in semantic policy evaluation
Pragmatic weight w_p	3.00	Reward for inspecting a site with high target probability
Epistemic weight w_e	1.50	Reward for anticipated entropy reduction
Repeated-site penalty w_a	1.00	Discourages immediate reinspection after negative evidence
Task relevance weight λ	0.75	Biases search towards objects that unlock more unfinished actions
State observation sensitivity	0.95	True-positive probability for kettle water inspection
State observation specificity	0.95	True-negative probability for kettle water inspection
Water-present goal	0.90	Threshold above which filling is unnecessary
Water-present boil precondition	0.85	Minimum belief required before boiling
Hot-water goal and pour precondition	0.90	Minimum belief that kettle water is hot
Tea-brewed goal and serve precondition	0.90	Minimum belief required before serving
Fill success probability	0.995	Posterior confidence after a successful fill action
Boil success probability	0.995	Posterior confidence that water is hot after boiling
Pour-to-mug water success	0.99	Posterior confidence that the mug contains water after pouring
Pour-to-mug heat success	0.97	Posterior confidence that mug water is hot after pouring

Parameter	Current value	Interpretation
Steep success probability	0.98	Posterior confidence that tea is brewed and teabag is steeped
Serve success probability	0.995	Posterior confidence that the mug has been served

References

[1] L. C. S. Berndt and A. D. Shaw, "Active Inference World Models: From Embodied Control to General-Purpose Adaptive Intelligence," Zenodo, 2026. doi:10.5281/zenodo.21325216.

[2] T. Parr and K. J. Friston, "Generalised free energy and active inference," *Biological Cybernetics*, vol. 113, pp. 495–513, 2019. doi:10.1007/s00422-019-00805-w.

[3] K. Friston, F. Rigoli, D. Ognibene, C. Mathys, T. Fitzgerald and G. Pezzulo, "Active inference and epistemic value," *Cognitive Neuroscience*, vol. 6, pp. 187–214, 2015. doi:10.1080/17588928.2015.1020053.

[4] R. Kaplan and K. J. Friston, "Planning and navigation as active inference," *Biological Cybernetics*, vol. 112, pp. 323–343, 2018. doi:10.1007/s00422-018-0753-2.

[5] M. Savva, A. Kadian, O. Maksymets, Y. Zhao, E. Wijmans, B. Jain, J. Straub, J. Liu, V. Koltun, J. Malik, D. Parikh and D. Batra, "Habitat: A Platform for Embodied AI Research," *Proceedings of the IEEE/CVF International Conference on Computer Vision*, 2019. arXiv:1904.01201.

[6] A. Szot, A. Clegg, E. Undersander, E. Wijmans, Y. Zhao, J. Turner, N. Maestre, M. Mukadam, D. Chaplot, O. Maksymets, A. Gokaslan, V. Vondrus, S. Dharur, F. Meier, W. Galuba, A. Chang, Z. Kira, V. Koltun, J. Malik, M. Savva and D. Batra, "Habitat 2.0: Training Home Assistants to Rearrange their Habitat," *Advances in Neural Information Processing Systems*, 2021. arXiv:2106.14405.

[7] J. Straub, T. Whelan, L. Ma, Y. Chen, E. Wijmans, S. Green, J. J. Engel, R. Mur-Artal, C. Ren, S. Verma, A. Clarkson, M. Yan, B. Budge, Y. Yan, X. Pan, J. Yon, Y. Zou, K. Leon, N. Carter, J. Briales, T. Gillingham, E. Mueggler, L. Pesqueira, M. Savva, D. Batra, H. M. Strasdat, R. De Nardi, M. Goesele, S. Lovegrove and R. Newcombe, "The Replica Dataset: A Digital Replica of Indoor Spaces," 2019. arXiv:1906.05797.