Master Thesis Projects on Interactive Reasoning Agents

Background and general expectations

Recent language-model agents combine neural models with memory, tools, executable code, search procedures, and explicit control mechanisms. A particularly useful setting for studying these systems is **interactive problem solving**, where an agent receives observations, chooses actions, observes their consequences, and gradually learns how an initially unfamiliar environment works.

ARC-AGI-3 is a representative benchmark in this direction. It consists of novel turn-based environments in which the agent is not given explicit rules or goals. Effective behaviour requires exploration, goal inference, modelling of environment dynamics, planning, and adaptation [1]. The official infrastructure supports programmatic agents and replayable experiments, making it suitable as one possible experimental testbed [2]. ARC-AGI-3 is deliberately difficult, however, and none of the projects below depends on solving the benchmark as a whole. Smaller subsets, controlled environments, or alternative benchmarks may be preferable when they make the intended experiment clearer.

These projects concern several related questions:

- how language models can cooperate with conventional algorithms and symbolic solvers;
- how agents can explicitly organise their problem-solving process;
- whether procedures learned while solving one task can be reused on later tasks;
- how these architectures behave when the underlying language model becomes smaller;
- how to build suitable software infrastructure for conducting such experiments reproducibly.

A Master's thesis in Information and Computer Science is expected to cover six months of full-time work. It need not introduce a fundamentally new research method. A successful thesis may reproduce an existing approach, apply an existing idea in a new setting, compare alternative implementations, or develop reusable experimental software. The expected outcome is a technically sound study documented in a thesis with a structure close to that of a research paper: motivation, related work, problem formulation, design, implementation, experimental methodology, results, discussion, and conclusions. Depending on the project, a substantial software artefact may be as important as the empirical results.

A useful working principle is to design each project around a **minimum result that is sufficient for a thesis**, with progressively more ambitious extensions. The student should have a working experimental pipeline well before the end of the project.

A typical schedule is:

Period	Expected checkpoint
Weeks 1–4	Study the problem and literature; install the selected benchmark; reproduce at least one existing baseline
Weeks 5–8	Define the experimental protocol; obtain a minimal end-to-end implementation
Weeks 9–14	Implement the main proposed system or software component
Weeks 15–18	Assess feasibility and freeze the scope; activate a fallback if necessary
Weeks 19–22	Run systematic experiments and analyse results
Weeks 23–26	Complete experiments, clean the implementation, and write the thesis

The checkpoint around months 3–4 is important. At that point, there should already be a defensible thesis path even if the original objective proves too difficult.

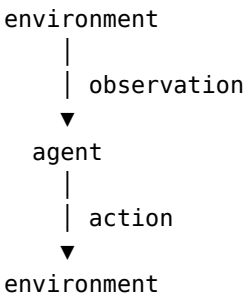
1. A Modular Experimental Platform for Interactive AI Agents

Objective

The goal is to develop a software platform for implementing, executing, inspecting, and comparing agents operating in interactive environments.

The project is primarily an **engineering thesis**. Its value lies in producing a clean experimental infrastructure that can subsequently support several different agent architectures and research projects.

The core abstraction should separate the environment from the agent:



The platform should make few assumptions about how the agent itself is implemented. One agent may be a direct LLM-based policy, another may use a state machine, another may invoke symbolic solvers, and another may not use a language model at all.

Expected activities

The student should design common interfaces for at least:

- environments;
- agents;
- model providers;
- external tools;
- execution traces;
- experiment configurations;
- evaluation metrics.

A minimal platform should support:

- reproducible experiment configurations;
- deterministic seeds where applicable;
- action and model-call budgets;
- structured logging;
- batch execution;
- persistent results;
- replay of completed trajectories.

Two execution modes are desirable.

The **headless mode** should support bulk experiments without visualisation. The **interactive mode** should allow a developer to inspect an ongoing or recorded execution, including observations, actions, model outputs, tool invocations, and internal state exposed by the agent.

ARC-AGI-3 is a natural first backend because official tooling already exposes an agent interface and benchmarking infrastructure [1,2]. The platform should wrap rather than reimplement that functionality.

Minimum viable thesis

A sufficient result would contain:

1. a generic environment API;
2. a generic agent API;
3. one interactive environment backend;
4. two simple reference agents;
5. a configurable experiment runner;
6. structured execution traces;
7. basic evaluation and replay facilities.

A sophisticated graphical interface is not required for the minimum result.

Possible extensions

If progress permits, the platform may add:

- an ARC-AGI-3 adapter;

- parallel execution;
- several language-model providers;
- token and monetary cost accounting;
- trace comparison;
- visual timeline inspection;
- inspection of explicit agent representations;
- checkpointing and experiment resumption;
- automatic generation of experimental reports;
- support for synthetic environments defined locally.

Main traps and fallback strategies

Building a GUI too early. A polished interface can consume most of the available time without improving the experimental core. Headless execution, logging, and replay should be implemented first. A simple Web or desktop trace viewer is sufficient if time remains.

Reimplementing existing benchmark infrastructure. ARC-AGI-3 already provides an SDK and an official benchmarking harness [2]. The thesis should concentrate on a reusable abstraction above existing environment APIs.

Designing abstractions without enough concrete agents. Generic interfaces often become unnecessarily complex if designed before real use cases exist. The student should implement two deliberately different agents early and evolve the abstraction from them.

Dependingence on remote services. An experimental framework that only works with one paid model API is difficult to reuse. At least one mock, deterministic, or locally executable agent should be available for testing.

Benchmark integration proving more difficult than expected. If ARC-AGI-3 integration becomes a bottleneck, the student can implement one or more small local environments and demonstrate that the architecture supports interactive agents independently of the selected benchmark.

2. Symbolic Search and Constraint Solving for LLM-Based Agents

Objective

The project investigates agents in which a language model handles interpretation and decomposition while conventional algorithms perform well-defined reasoning operations.

Language models can often identify useful structure in a problem while remaining unreliable at exact symbolic computation. Several previous approaches therefore combine language models with external tools, modules, or executable programs [3–5]. The thesis studies this idea in interactive problems, where symbolic computation may also support planning over an inferred environment model.

Candidate tools include:

- A* or another graph-search algorithm;
- Z3 or another SMT solver;
- a Python interpreter;
- a simple simulator or planning engine.

The goal is not to develop new search or constraint-solving algorithms. The relevant question is whether an agent can construct the information required by such algorithms and use their output effectively.

A representative architecture is:

```
observation
    ↓
language model
    ↓
structured problem representation
    ↓
search / solver / executable model
    ↓
candidate plan or deduction
    ↓
agent action
```

Expected activities

The student should first implement a simple language-model agent that interacts directly with the chosen environment.

A symbolic operator should then be integrated. For example, an agent using A* may need to determine:

- what constitutes a state;
- which actions generate transitions;
- which state is currently observed;
- what goal should be reached;
- which states are equivalent;
- possibly which heuristic should be used.

Similarly, an agent using Z3 must translate observations and hypotheses into variables and constraints.

The experimental study should compare at least a direct agent with a tool-assisted variant.

Minimum viable thesis

A satisfactory minimum scope is:

- one interactive benchmark or small family of environments;
- one language model;
- one symbolic technique;
- one direct baseline;

- one tool-assisted architecture;
- quantitative and qualitative comparison of their behaviour.

The representation supplied to the symbolic tool may initially follow a manually designed schema. Autonomous discovery of the schema is an extension rather than a prerequisite.

Possible extensions

More ambitious versions may investigate:

- several symbolic operators;
- automatic selection of the appropriate operator;
- dynamically generated representations;
- executable world models;
- validation of solver predictions against subsequent observations;
- correction of an inferred model after contradictory evidence.

Main traps and fallback strategies

The benchmark may simply be too difficult. ARC-AGI-3 deliberately requires several capabilities simultaneously [1]. If the complete task overwhelms the experimental system, the student should select a small subset of environments, impose simplifying assumptions, or construct controlled interactive tasks where the intended use of search or constraints can be studied cleanly.

The hard problem may be representation construction rather than symbolic reasoning. A solver cannot compensate for an incorrectly encoded problem. If autonomous representation construction proves unreliable, progressively constrain the problem:

```
agent invents representation
        ↓
agent populates predefined representation
        ↓
agent populates partially pre-filled representation
        ↓
representation supplied by the experimenter
```

The lower levels remain useful for studying the effect of symbolic reasoning.

Frontier model access may be expensive. Interactive agents can generate many API calls per episode. Before starting large experiments, the student should identify a sustainable source of API credits, such as institutional access, research or educational credits, or an appropriate free allocation. Experiments should not depend on the student personally funding repeated frontier-model calls.

API behaviour and model availability may change during the thesis. Model access should therefore be isolated behind a common interface and experiments should log the exact model version and configuration.

The project may expand into too many tools. One well-studied symbolic operator is sufficient. Comparing A*, Z3, theorem proving, program synthesis, and planning within one thesis would normally be excessive.

3. Learning and Reusing Problem-Solving Skills

Objective

This thesis studies whether an interactive agent can retain useful information from solved tasks and exploit it when encountering related problems.

Existing agents have explored several forms of experience reuse. Reflexion stores linguistic feedback from previous attempts [6], while Voyager maintains a library of executable skills that can later be retrieved and composed [7]. This project adapts the general idea to interactive reasoning tasks.

A useful notion of *skill* may initially be pragmatic. Examples include:

- a natural-language description of a previously discovered rule;
- a reusable problem-solving procedure;
- a parameterised representation;
- an executable program;
- a solver template;
- a sequence of actions with applicability conditions.

The basic process is:

```
solve task
    ↓
extract reusable information
    ↓
store skill
    ↓
encounter related task
    ↓
retrieve and adapt skill
    ↓
solve task
```

Expected activities

The student should construct a family of related tasks or identify a benchmark in which experience from earlier tasks may plausibly help on later ones.

At least two memory strategies should be compared. A simple experiment might compare:

- no reuse;
- retrieval of complete previous trajectories;
- retrieval of concise task summaries;
- retrieval of structured or executable skills.

The central measurement is whether previous experience reduces the effort required on later tasks.

Relevant measures include:

- task success;
- number of interactions;
- number of model calls;
- token usage;
- execution time;
- proportion of retrieved skills that are actually useful.

Minimum viable thesis

The simplest defensible version does **not** require automatic skill discovery.

The student may define a structured skill format manually and study whether storing and retrieving such artefacts helps compared with storing raw histories.

This already permits useful experiments on representation, retrieval, and transfer.

Possible extensions

Extensions include:

- automatic extraction of skills from successful trajectories;
- automatic estimation of applicability conditions;
- executable skills rather than textual summaries;
- skill composition;
- adaptation of a retrieved skill to a slightly different problem;
- removal or revision of skills that repeatedly fail;
- transfer between different environments sharing some structural regularity.

Main traps and fallback strategies

There may be no meaningful transfer between benchmark tasks. If tasks are deliberately unrelated, memory cannot reasonably help. The experimental dataset must contain a controlled notion of relatedness. Synthetic task families may therefore be more useful than an existing benchmark.

A retrieval system can appear useful merely by leaking task-specific answers. Training and evaluation tasks should differ enough that successful transfer requires reuse of a general procedure rather than retrieval of the final solution.

Automatic skill induction may consume the entire project. The fallback is to provide the skill representation manually and focus on storage, retrieval, adaptation, and evaluation.

It may be difficult to distinguish useful skills from larger prompts. Experiments should include a raw-history or context-only baseline. Otherwise, a richer skill representation cannot be distinguished from simply providing the model with more information.

Frontier models may be required to obtain reasonable behaviour. As in the previous project, repeated API usage can become costly. Model access and expected experiment volume should be checked before selecting the final benchmark.

4. Robustness of Agent Architectures Across Language-Model Scales

Objective

Many proposed agent architectures are demonstrated using strong proprietary models. It is less clear which parts of their behaviour depend on the underlying model and which can be preserved by explicit reasoning structure, external tools, or deterministic control.

This thesis studies how one or more agent architectures behave when the underlying language model is progressively replaced by smaller models.

The main experiment has two dimensions:

[.]

For example:

Model	Direct agent	Structured/tool-assisted agent
strong model	experiment	experiment
medium model	experiment	experiment
small model	experiment	experiment

The purpose is not necessarily to demonstrate that smaller models can match frontier models. More useful results may identify which components fail first as the model becomes less capable.

SELF-DISCOVER provides one relevant example of explicit reasoning structures being transferable across model families [8]. More generally, program-aided and modular systems motivate separating semantic interpretation from computations that can be performed reliably by external components [4,5].

Expected activities

The student should begin from an existing agent implementation, either produced previously or reproduced from the literature.

Several models should then be evaluated under approximately comparable conditions. Useful diagnostics include:

- task success;
- invalid actions;
- failures in structured-output generation;
- incorrect state extraction;
- failed tool calls;
- planning failures;
- inability to recover after errors;
- token and computational cost.

The most interesting analysis may decompose a complete agent trajectory into sub-capabilities such as:

```
observation interpretation
        ↓
state construction
        ↓
hypothesis generation
        ↓
tool selection
        ↓
formal encoding
        ↓
planning
        ↓
execution
```

Minimum viable thesis

A sufficient project may use:

- one agent architecture;
- three model sizes or capability levels;
- a manageable task subset;
- detailed failure analysis.

The architecture itself need not be novel.

Possible extensions

Possible extensions include:

- comparison between direct and structured agents;
- explicit measurement of representation quality;
- quantised versus full-precision local models;
- experiments controlling inference-time compute;
- replacing selected stages of a large-model agent with smaller specialist models;
- studying whether symbolic tools reduce dependence on model scale.

Main traps and fallback strategies

Running small models is not automatically free. Open weights remove API charges but not compute requirements. Before committing to the experiment, the student should verify access to suitable GPUs through university infrastructure, research servers, free cloud allocations, or machines capable of running quantised models. The selected models should reflect the hardware actually available.

Large models and small models may expose different interfaces. Structured output, tool calling, vision support, context length, and reasoning controls may differ substantially. The architecture must therefore avoid assuming proprietary API features that cannot be reproduced locally.

The smallest models may fail almost completely on the full benchmark. This is a likely outcome on demanding interactive tasks. The fallback is to evaluate individual stages separately—for example state extraction, rule induction, tool selection, or constraint generation—rather than insist on full end-to-end success.

Model size alone is an imperfect independent variable. Different model families differ in training data, architecture, modality support, and post-training. The thesis should describe results as comparisons between selected models or capability levels rather than claim a universal law of parameter scaling.

Experiments may become computationally excessive. A factorial comparison across many agents, models, benchmarks, seeds, and prompts grows quickly. A smaller experiment with enough repeated runs is preferable to a large but statistically weak matrix.

5. Deterministic Control Architectures for Language-Model Agents

Objective

Most LLM-based agents give the language model considerable control over the reasoning trajectory. ReAct, for example, interleaves reasoning with actions selected by the model [3]. Other methods organise reasoning as trees or graphs [9,10].

This thesis explores a different engineering choice: the overall reasoning procedure is explicitly implemented by the developer, while the language model is invoked only for bounded operations inside that procedure.

An example controller might contain states such as:

```
OBSERVE
   ↓
INTERPRET
   ↓
GENERATE HYPOTHESES
   ↓
SELECT EXPERIMENT
   ↓
ACT
   ↓
UPDATE MODEL
   ↓
PLAN
   ↓
EXECUTE
   ↓
VERIFY
```

Each state has explicit input and output types, and transitions between states are controlled by code rather than unrestricted model generation.

This architecture may be implemented as a finite-state machine, hierarchical state machine, workflow graph, or similar deterministic controller.

Expected activities

The student should first reproduce a simple direct or ReAct-style baseline [3].

A second agent should then implement an explicit problem-solving cycle. Individual states may use the language model for tasks such as:

- interpreting an observation;
- generating hypotheses;
- summarising accumulated evidence;
- proposing candidate goals;
- encoding a problem for a solver.

Other states can be entirely deterministic.

The thesis should study whether explicit control affects reliability, efficiency, interpretability, or dependence on model capability.

Minimum viable thesis

A valid minimum system may contain only a small cycle:

```
OBSERVE → REASON → ACT → VERIFY
```

with persistent structured state between iterations.

Comparison against a less structured baseline on a selected task set is sufficient.

Possible extensions

The controller can gradually be enriched with:

- explicit hypothesis management;
- separate exploration and exploitation modes;
- world-model construction;
- confidence estimates;
- symbolic solvers;
- backtracking;
- explicit diagnosis of failed predictions;
- different state transition policies.

At the ambitious end, the project could compare fixed reasoning structures against dynamically generated reasoning graphs. Tree-of-Thoughts, Graph-of-Thoughts, and SELF-DISCOVER provide useful points of comparison [8–10].

Main traps and fallback strategies

The state machine can become an arbitrary collection of prompts. The design should assign a clear semantic responsibility and data contract to each state. Otherwise, the architecture is difficult to analyse or compare.

Too many states can make the system brittle. Start from the smallest useful reasoning loop. Additional states should be justified by an observed failure mode.

A deterministic controller does not guarantee deterministic behaviour. The language model inside each state remains stochastic unless constrained. Experiments should record seeds and sampling parameters where supported and repeat runs where variability matters.

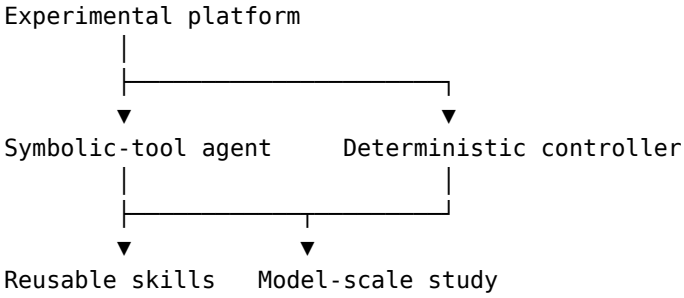
ARC-AGI-3 may be too demanding for the first implementation. A simpler environment should be used initially. The same controller can later be tested on a sample of harder tasks.

Small-model execution may become a hardware problem. If the project specifically targets small local models, compute availability should be established before defining the model set. A practical fallback is to develop the architecture using inexpensive hosted models and perform the final small-model comparison only on a representative task subset.

Frontier-model APIs can also become a cost bottleneck. If a strong model is needed as a reference baseline, access to suitable credits should be arranged early. The thesis should remain viable with a smaller number of frontier-model evaluation runs.

Relationship among the projects

The projects can be carried out independently, but they naturally support an incremental programme:



The experimental platform provides reusable infrastructure. The symbolic-tool and deterministic-controller projects provide two distinct agent architectures. Subsequent theses can investigate transfer and model scaling without first rebuilding the complete experimental stack.

Continuity should remain an advantage rather than a prerequisite. If previous software is unavailable, incomplete, or unsuitable, every thesis should retain a reduced standalone path.

Across the projects, a common experiment format would be useful. At minimum, executions should record:

- benchmark and task identifier;
- complete agent configuration;
- exact model and model configuration;
- observations and actions;
- model calls;
- external tool calls;

- token usage;
- computational or monetary cost when available;
- final task outcome.

This will make results from successive theses easier to reproduce and compare.

Common practical risks

A few constraints affect most of these projects and should be checked before assigning a thesis.

Access to strong models

Repeated calls to frontier models may cost more than is reasonable for a Master's project, particularly for interactive benchmarks where a single episode can require many model invocations. A project relying on such models should start only after identifying a sustainable source of API access or credits. The software should make it possible to switch providers and models without rewriting the agent.

Access to GPUs for open models

Open-weight models avoid commercial API costs but require computation. Model selection should follow available hardware rather than precede it. Quantisation can reduce memory requirements, but systematic experiments may still require substantial GPU time.

Benchmark difficulty

ARC-AGI-3 is intended to expose limitations of current agents, and complete benchmark performance is not an appropriate success criterion for a six-month thesis [1]. Valid strategies include:

- selecting a representative task subset;
- focusing on particular environment families;
- simplifying the observation or action interface;
- constructing smaller diagnostic environments;
- replacing ARC-AGI-3 with another interactive benchmark when this makes the intended experiment more informative.

Failure to solve a difficult benchmark is not itself informative unless the experimental setup allows the causes of failure to be analysed.

Experimental cost

Interactive agents can consume large numbers of tokens and actions. The student should estimate the cost of a complete experiment matrix before running it. Development runs should use reduced budgets, small task subsets, and inexpensive models.

Reproducibility

Hosted models change, API behaviour evolves, and stochastic agents can vary substantially between runs. Model identifiers, prompts, configurations, source-code revisions, benchmark versions, and random seeds should therefore be stored with the results.

References

[1] ARC Prize Foundation. **ARC-AGI-3: A New Challenge for Frontier Agentic Intelligence.** 2026. ARC-AGI-3 introduces interactive, abstract environments requiring exploration, goal inference, world-model formation, planning, and adaptation. :chatgpt-content-reference{index="0"}

[2] ARC Prize Foundation. **ARC-AGI-3 Benchmarking.** Official open-source benchmarking infrastructure, 2026. It provides environment access, model-provider adapters, experiment execution, and support for public ARC-AGI-3 games. :chatgpt-content-reference{index="1"}

[3] Yao, S., Zhao, J., Yu, D., Du, N., Shafran, I., Narasimhan, K., and Cao, Y. **ReAct: Synergizing Reasoning and Acting in Language Models.** ICLR, 2023. :chatgpt-content-reference{index="2"}

[4] Karpas, E., et al. **MRKL Systems: A Modular, Neuro-Symbolic Architecture That Combines Large Language Models, External Knowledge Sources and Discrete Reasoning.** arXiv:2205.00445, 2022. :chatgpt-content-reference{index="3"}

[5] Gao, L., Madaan, A., Zhou, S., Alon, U., Liu, P., Yang, Y., Callan, J., and Neubig, G. **PAL: Program-Aided Language Models.** ICML, 2023. :chatgpt-content-reference{index="4"}

[6] Shinn, N., Cassano, F., Berman, E., Gopinath, A., Narasimhan, K., and Yao, S. **Reflexion: Language Agents with Verbal Reinforcement Learning.** NeurIPS, 2023. :chatgpt-content-reference{index="5"}

[7] Wang, G., Xie, Y., Jiang, Y., Mandlekar, A., Xiao, C., Zhu, Y., Fan, L., and Anandkumar, A. **Voyager: An Open-Ended Embodied Agent with Large Language Models.** 2023. Voyager introduces an executable skill library with retrieval and compositional reuse. :chatgpt-content-reference{index="6"}

[8] Zhou, P., et al. **SELF-DISCOVER: Large Language Models Self-Compose Reasoning Structures.** NeurIPS, 2024. :chatgpt-content-reference{index="7"}

[9] Yao, S., Yu, D., Zhao, J., Shafran, I., Griffiths, T. L., Cao, Y., and Narasimhan, K. **Tree of Thoughts: Deliberate Problem Solving with Large Language Models.** NeurIPS, 2023.

[10] Besta, M., et al. **Graph of Thoughts: Solving Elaborate Problems with Large Language Models.** AAAI, 2024. :chatgpt-content-reference{index="8"}