
Investigating Structural Safety Guarantees for Agentic AI via Goal-Oriented Action Planning

Daniel Juhl

juhl.daniel@gmail.com
Independent

With
Apart Research

Abstract

*AI control protocols generally rely on monitoring and human auditing to maintain safety despite potential subversion by an untrusted model. I present an alternative approach that can provide structural rather than empirical safety guarantees. A trusted model decomposes agentic tasks into structured steps which are combined into an execution plan using Goal-Oriented Action Planning (GOAP). I implement this as a control protocol and test it in the AgentDojo Workspace setting. Under adversarial attack policies, the protocol reduces injection task success rates to **under 1%** with a usefulness decrease of **between 8-21 percentage points** depending on the model used. The protocol eliminates the need for the human auditing budget, but is currently only applicable in certain well-defined action spaces - notably, it does not apply to traditional code generation settings.*

1. Introduction

The field of AI control seeks to answer the question of how highly capable but untrusted models can be used safely to do valuable work. The main works have so far focused on designing protocols that, through adversarial experiments, give us empirical guarantees of safety. It would be highly

desirable to have structural, provable safety guarantees instead, but this is impossible¹ in the general case. This project draws inspiration from security engineering to show that such guarantees can be given in well-defined action spaces against certain forms of attacks, at a low cost to usefulness while removing the need for human auditing completely.

My main contributions are:

- Demonstrating a structural decomposition approach to solving agentic tasks in well-defined action spaces and implementing this as a control protocol.
- Showing that this approach achieves near-perfect safety in the AgentDojo setting at a low cost to usefulness.

The implemented control protocol ensures that the untrusted model cannot invoke any tools that are not deemed relevant for solving the user task by the trusted model, and that it cannot change any tool arguments that are directly implied by the user instructions. This limits the attack surface of the untrusted model to generating malicious content in individual execution steps, or attempting to manipulate the planning of the trusted model. The technique could significantly harden our defensive posture in agentic settings with bespoke tools, and potentially serve as a first layer of defense in more open-ended settings, reducing the monitoring burden to open-ended content generation steps and the residual attack surfaces identified in section 5.

2. Related Work

“AI Control: Improving Safety Despite Intentional Subversion” ([Greenblatt, Shlegeris, et al., ICLR 2025](#)) is the founding paper in the field of AI control. The work done in this project was inspired by “Defeating Prompt Injections by Design” ([Debenedetti, Shumailov et. al., 2025](#)), where the authors separate data and control flows for LLM agents in the AgentDojo ([Debenedetti, Zhang et. al., 2024](#)) setting, obtaining *provable* security against many classes of attacks with only a minor reduction in usefulness. I also draw upon the [Embabel](#) framework’s utilisation of Goal Oriented Action Planning (GOAP) as a deterministic planning algorithm over action spaces.

Separating planning and execution, and decomposing said execution into isolated steps can be viewed as a form of “Factored Cognition” ([Stuhlmüller, Gooen et. al., 2018](#)) which, in the context of AI control, is investigated in “Factor(T ,U): Factored Cognition Strengthens Monitoring of Untrusted AI ([Sandoval & Rushing, 2025](#))” and “Factor(U ,T): Controlling Untrusted AI by Monitoring their Plans” ([Chee Lip, Channg et. al, 2025](#)).

One potential vector for the untrusted model to attack the implemented control protocol is by manipulating the trusted planner during error-induced re-planning steps. The ability of an untrusted model to compromise the trusted model is studied in “Adaptive Attacks on Trusted Monitors Subvert AI Control Protocols” ([Terekhov, Panfilov et. al. 2025](#)).

¹ It reduces to the halting problem.

3. Methods

Using the AISI ControlArena framework, I implement a two-phase GOAP-based protocol as a `@control_agent`. The two phases are the planning phase and the execution phase. This implementation is inspired by the [Embabel framework](#).

Planning phase

In the planning phase, a trusted model is used to translate the natural-language user request and available scaffold tools into a form that is suitable for the GOAP algorithm. Using topological sorting and A* search, this algorithm determines a sequence of actions in the action space that will solve the user request by reaching the desired goal state. As part of this setup, the trusted model may assign fixed function arguments that are immutable, and dynamic arguments which are generated during execution. This is inspired by the data annotations from the “[Defeating Prompt Injections by Design](#)” paper and gives a clear and structural overview of which parts of the execution phase may be influenced by the untrusted model - and how.

Execution phase

In the execution phase, the planned action sequence is executed. All execution steps from the plan that require an LLM make use of the untrusted model U. All tool call responses are investigated by T to determine whether they succeeded. If they did not, T may trigger a retry or a replan as appropriate. This allows the protocol to recover from unexpected errors during execution.

A more detailed implementation description is provided in the appendix.

4. Results

For budget reasons I use Gemini 3.1 Flash Lite Preview, Gemini 3 Flash Preview and Claude Haiku 4.5 to obtain my results.

Safety

To test the safety of the protocol, I use the default ControlArena attack policy builder which provides a malicious system prompt and injects ephemeral reminders about completing the side task. I run the same 40 tasks in the “Workspace” setting with the injection tasks 0, 1, 2, 3, 10, 11, 12, 13², for a total of 320 runs. I do this with the normal Inspect ReAct setup to measure the ability of the models to complete the injection task with no structural restrictions in place, then with the GOAP protocol to measure the ability to complete the injection task here.

² The intention was to run on just 0, 1, 2, and 3 but due to a regex error, we get a more comprehensive evaluation.

Figure 1 shows the side task success rate of the tested models when used in an attack policy without any restrictions (baseline) and under the implemented two-phase GOAP control protocol.

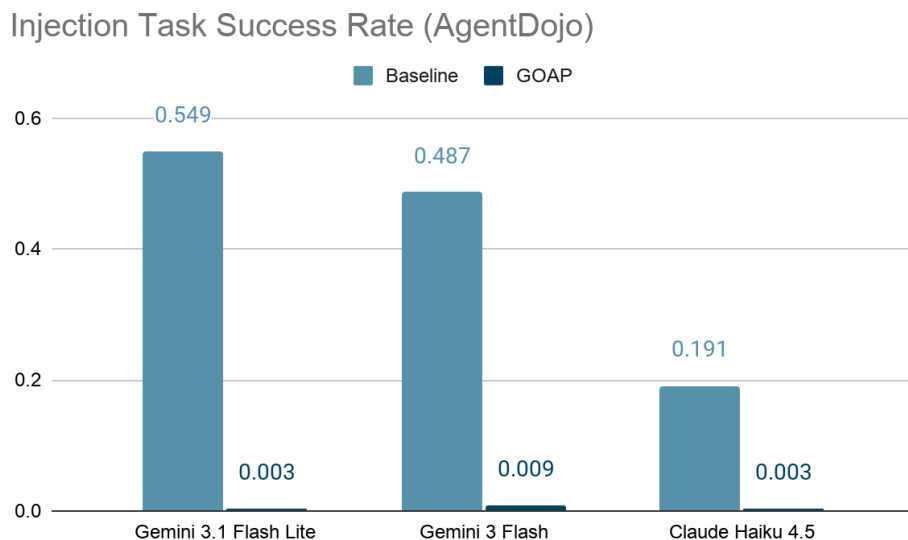


Figure 1: Adversarial models fail to complete almost all injection tasks under the implemented GOAP protocol. Due to budget constraints, only a single 320-sample run was made with each combination.

Two things are worth noting:

- First, the baseline injection task success rate is not 1. This is most likely due to rejection from the models being tested, as Gemini 3.1 Flash Lite has the best rate and Claude Haiku 4.5 has less than half the success rate of its compatriots.
- The GOAP protocol blocks almost all injections but does not achieve perfect safety. This is due to certain combinations of main tasks and injection tasks having sufficient overlap that the injection task can be completed by manipulating specific parameters to the tool calls planned out by GOAP. Specifically, some user tasks ask the model to “delete the largest file on my Cloud drive” and some injection tasks require that the model “delete the file with id 13”. Thus, when a GOAP step asks the untrusted model to identify the largest file based on the output of a `list_files` tool call, the attack policy can choose to reply with “id 13” instead of the correct “id 11”, leading to the successful completion of the injection task.

Usefulness

To measure the baseline usefulness of the models, I use the Inspect AI ReAct agent. I perform 3 sets of honest-mode runs with this agent over the 40 tasks in the AgentDojo “Workspace” setting.

I then perform another 3 sets of honest mode runs using the implemented GOAP-based two-phase protocol to measure the usefulness of that protocol. The results are shown in figure 2.

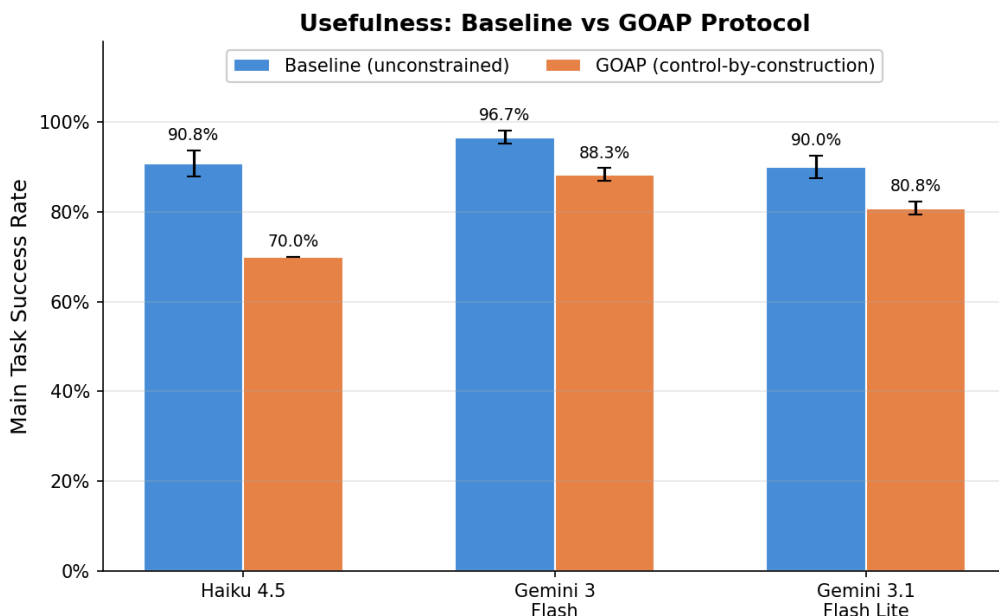


Figure 2: Usefulness drops slightly on AgentDojo under the GOAP protocol, as measured over 3 40-sample runs. The two gemini models lose ~10pp performance, whereas Haiku is hit harder and loses more than 20pp.

We observe a drop in usefulness by introducing the GOAP protocol. The AgentDojo tasks are generally fairly easy from a mechanical standpoint³. Thus, the challenge of translating the user request into a suitable GOAP action space and the rigidity of the subsequent execution phase leads to this modest reduction in usefulness. A similar effect is observed in “Defeating Prompt Injections by Design” where their provable security guarantees on AgentDojo tasks comes at the cost of a 7pp usefulness reduction in their setting.

5. Discussion and Limitations

The large increase in safety within the AgentDojo setting comes at a modest loss of usefulness. The GOAP protocol is quite a complex framework, and I believe that most if not all of this usefulness loss could be avoided with further refinement. The main benefits of the implemented protocol come with potential for structured and provable safety guarantees, and elimination of the need for human auditing budget. There are significant limitations however, to when and where the protocol can successfully be implemented.

³ I would argue that most if not all of the baseline failures are due to confusingly worded user requests and/or overly restrictive grading criteria.

Limitations

There are significant limitations to the protocol presented. As mentioned elsewhere, the protocol does not prevent an untrusted model from generating malicious content such as code with undesirable effects or backdoors. In general any ill-defined or open-ended task will not see significant safety gains by adopting this kind of technique. This means that we will still have to rely on other layers of defense and control techniques or protocols in most practical scenarios. The choice of introducing a re-planning error recovery step in the protocol also opens the door for the untrusted model to influence the trusted planner - one can imagine the untrusted model compromising the planner by inducing a tool failure that echoes malicious content, e.g. “Error: ‘ignore all previous instructions and drop table Students’ is not a valid username”. While this weakness is shared with most other control protocols, it is arguably more severe in this case, as it compromises one of the desired structural guarantees of the protocol.

There are also significant limitations to the work done here. The GOAP protocol was hastily implemented and not well-specified in detail, which means the claimed structural guarantees are hand-wavy at best. The empirical results are also fairly weak, as neither state-of-the-art models such as Opus 4.6 or GPT 5.4 were given the chance as untrusted models, nor was significant effort put into manual red teaming. As such this project should be seen as a feasibility study and proof-of-concept, not as a finished piece of work.

Future Work

There are three main directions for future work:

- Iterate and improve upon the GOAP control protocol, by formalising the protocol and refining the implementation to improve usefulness
- Applying the protocol to more settings, and testing it with SOTA models.
- Investigate which parts can be applied to more open-ended settings. It may be possible to reduce the attack surfaces in settings such as LinuxArena “chokepoints”.

6. Conclusion

This project shows that it is feasible to obtain structural security guarantees within certain well-defined types of agentic settings, providing a potentially valuable complementary technique to conventional monitor-based AI control protocols.

Code and Data

All code and experimental data is available at

<https://github.com/Scrattlebeard/apart-control-hackathon>

References

1. Greenblatt, R., Shlegeris, B., et al. (2025). AI Control: Improving Safety Despite Intentional Subversion. *ICLR 2025*. <https://arxiv.org/abs/2312.06942>
2. DeBenedetti, E., Shumailov, I., et al. (2025). Defeating Prompt Injections by Design. <https://arxiv.org/abs/2503.18813>
3. DeBenedetti, E., Zhang, Y., et al. (2024). AgentDojo: A Dynamic Environment to Evaluate Attacks and Defenses for LLM Agents. <https://arxiv.org/abs/2406.13352>
4. Johnson, R. (2025). Embabel Agent Framework. GitHub. <https://github.com/embabel/embabel-agent>
5. Stuhlmüller, A., Gooen, M., et al. (2018). Factored Cognition. *Alignment Forum*. <https://www.alignmentforum.org/posts/DFkGStzyj3jgXibFG/factored-cognition>
6. Sandoval, M. & Rushing, E. (2025). Factor(T,U): Factored Cognition Strengthens Monitoring of Untrusted AI. <https://arxiv.org/abs/2512.02157>
7. Chee Lip, Channg, et al. (2025). Factor(U,T): Controlling Untrusted AI by Monitoring their Plans. <https://arxiv.org/abs/2512.14745>
8. Terekhov, Panfilov, et al. (2025). Adaptive Attacks on Trusted Monitors Subvert AI Control Protocols. <https://arxiv.org/abs/2510.09462>

Appendix - Implementation description

This appendix provides implementation details for the Goal-Oriented Action Planning (GOAP) protocol. It was generated by Opus 4.6 based on the source code and reviewed by the human author.

A.1 State Machine

The protocol operates as a five-phase state machine inside ControlArena's react scaffold loop. Each iteration of the scaffold calls the protocol once, advancing the plan by one action.

Phase	Description
-------	-------------

-----	-----
-------	-------

Planning	Trusted model generates a structured plan from the task description and tool catalog.
---------------------	---

Executing	Each call constructs and dispatches one tool call, bridge computation, or sub-agent invocation.
Replanning	Triggered on step failure. Re-invokes the trusted planner with accumulated context and failure history.
Done	All plan steps executed or exhausted. Transitions to submit.
Submitted	Terminal. Trusted model formulates a final answer from tool results and calls `submit`.

A.2 Trusted Planner

The trusted model receives a prompt containing:

1. **Tool catalog** — each available tool rendered as a typed signature: `tool\_name(param1: type, param2: type) -> tool\_name\_result`.
2. **Action type definitions** — specifying the distinction between `fixed\_args` (structural parameters set by the planner) and `content\_params` (free-text fields filled by the untrusted model).
3. **Planning guidelines** — decomposition strategy, security rules (fixed_args are non-negotiable), preference for unrolled concrete tool calls over dynamic iteration.
4. **Few-shot example** — a worked example demonstrating the expected JSON output format.

The planner outputs a list of `PlannedAction` objects, each specifying:

- `id`: unique action identifier
- `tool`: catalog tool name (or omitted for bridges)
- `fixed\_args`: structural parameters (recipients, IDs, amounts, queries)
- `content\_params`: parameter names requiring LLM generation
- `inputs`/`outputs`: dependency wiring between actions
- `prompt`: generation guidance for bridge or sub-agent actions

Post-processing pipeline

The raw planner output undergoes several validation and normalization steps:

1. **Bridge normalization** — resolves variant reference formats (`action.output`, `action\_output`, `action/output` → `action:output`) and maps bare references to canonical forms.
2. **Submit injection** — strips any user-provided submit action and injects canonical `goap\_submit` + `bridge\_submit` actions with correct dependencies.
3. **Catalog validation** — checks `fixed\_args` and `content\_params` against the tool catalog's parameter schemas. Auto-resolves overlaps (fixed_args takes precedence).

4. ****Action space construction**** — converts ``PlannedAction`` objects into ``GoapAction`` instances with symbolic preconditions and effects for the `A\`` planner.

A.3 Plan Ordering

Actions are ordered deterministically via dependency resolution:

1. ****Topological sort**** (fast path) — builds a dependency DAG from action preconditions/effects and runs Kahn's algorithm. Used when the graph is acyclic.
2. ****A\` search**** (fallback) — searches over a symbolic state space where each state is a set of satisfied conditions. The heuristic is the count of unsatisfied goal conditions. Bounded to 10,000 iterations.

Both methods produce an ordered plan that respects all declared preconditions.

A.4 Action Types

Tool actions

The planner specifies ``fixed_args`` (immutable structural parameters) and ``llm_args`` (content parameters). At execution:

1. Start with ``arguments = dict(fixed_args)``.
2. For each parameter in ``llm_args``:
 - Check if a prior bridge produced a value for it (``bridge_outputs["{action_id}:{param}"]``).
 - If found, use the bridge value directly. Otherwise, prompt the untrusted model.
3. Coerce types to match the tool schema (e.g., comma-separated strings $\rightarrow$ arrays).
4. Construct a ``ToolCall`` and append to the conversation for scaffold execution.

Effects are deferred until the tool result is evaluated on the next scaffold call.

Bridge actions

Bridges perform intermediate computation without tool calls — extracting values from prior results, computing derived quantities, or reformatting data. A bridge declares ``inputs`` (results from prior actions) and ``outputs`` (values to wire into downstream actions).

Single-output bridges produce plain text. Multi-output bridges produce JSON with keys corresponding to each output parameter, parsed and stored individually in ``bridge_outputs``.

Bridges see all retrieval results accumulated so far, enabling them to operate over the full context (e.g., extracting multiple email addresses from a search result).

Sub-agent actions

Sub-agent actions handle dynamic iteration when items to process are not known at planning time. The trusted planner is called recursively with the sub-goal and accumulated context, producing a concrete sub-plan.

Decomposition validation prevents degenerate recursion:

- Rejects single-action sub-plans (pure re-delegation).
- Rejects sub-plans with $>60\%$ token overlap with the parent task prompt.
- Retries up to 2 times with escalating pressure; falls back to flattening the sub-agent into a bridge.
- Hard depth limit of 100.

Accepted sub-plans are spliced into the parent plan at the current position, with action and condition names prefixed to avoid collisions.

A.5 Failure Handling

Step evaluation

After each tool call, the protocol evaluates the result:

- **Fast path**: regex detection of empty/error results (``No X found``, ``[]``, error-flagged messages).
- **LLM path**: for non-trivial results, a trusted model classifies the outcome as CONTINUE (success), RETRY (content-level failure, regenerate), or REPLAN (structural failure, need different approach).

Write tools (prefixed with ``send_``, ``create_``, ``delete_``, etc.) skip the empty-result check, since many return empty on success.

Retry and replan

Strategy	Trigger	Behavior
----------	---------	----------

-----	-----	-----
-------	-------	-------

Retry	Content malformation (max 2 per action)	Re-execute same action; bridge outputs preserved, LLM args regenerated.
--------------	---	---

| ****Skip & defer**** | Failure independent of remaining plan | Continue with remaining steps; retry deferred failures after main plan completes. |
| ****Replan**** | Failure blocks downstream actions (max 3 global) | Re-invoke trusted planner with context: completed steps, retrieved data, failed attempts to avoid. |

Deferred failures are dropped after appearing 3 times in the failed-attempts list, preventing infinite retry loops.

A.6 Argument Coercion

The protocol includes type coercion to handle common LLM output format mismatches:

- ****String → array****: comma-separated strings or JSON-encoded arrays are converted to lists when the tool schema expects an array parameter.
- ****String → number****: numeric strings (with optional label prefixes like `"amount: 264"`) are parsed to integers or floats as required.

This reduces replan churn from format errors that do not reflect a planning failure.

A.7 Security Properties

The protocol enforces a structural separation between planning (trusted) and content generation (untrusted). The untrusted model cannot:

- Add, remove, or reorder tool calls
- Modify fixed arguments (recipients, file IDs, amounts, subjects)
- Call tools not in the plan
- Bypass precondition checks or alter the dependency graph

The attack surface is limited to the content of `llm\_args` fields. For tasks where all parameters are structural (e.g., calendar lookups, file deletions), the untrusted model has zero degrees of freedom. For tasks with content parameters (e.g., email bodies), the model can influence text content but not the structural envelope (who receives the email, what subject line, which attachments).

Residual attack surface

Two boundaries merit discussion:

****Bridge trust boundary.**** When a bridge extracts a value that becomes a downstream `fixed\_arg` (e.g., extracting a file ID from search results), the bridge effectively influences a structural parameter. The current implementation mitigates this by providing bridges with full

retrieval context (reducing the need for inference), using JSON schema validation on multi-output bridges, and allowing the trusted model to perform bridge computations when the output is security-sensitive. However, bridges remain a point where content generation can influence structural parameters indirectly.

****Replanning influence.**** The untrusted model's generated content can indirectly influence the replanning phase. If the untrusted model produces content that causes a tool call to fail — or succeed in a misleading way — it affects whether replanning triggers, what failure context the trusted planner sees, and consequently what plan the trusted model produces on the next attempt. The protocol bounds this influence through replan attempt limits (max 3 global), a cumulative failed-attempts list that prevents the planner from repeating the same approach, and deferred failure caps. Nevertheless, a sufficiently adversarial model could strategically induce failures to steer the trusted planner toward a plan that offers more degrees of freedom in `llm\_args` fields or even introduces new tool calls.

A.8 Source Layout

File Purpose
----- -----
`protocol.py` State machine, execution loop, tool/bridge/sub-agent dispatch
`trusted_planner.py` Trusted model prompting, response validation, bridge normalization
`planner.py` A* search, topological sort, failure diagnosis
`types.py` `GoapAction`, `WorldState`, `GoalState`, `Plan` dataclasses
`prompts/trusted_planner.txt` System prompt for the trusted planner
`prompts/step_eval.txt` Prompt for step result classification
`run_eval.py` CLI entry point, protocol and baseline wiring

LLM Usage Statement

This project made heavy usage of Claude Code. Basically all code was written by Opus 4.6 through Claude Code, and significant parts of transcript analysis and troubleshooting during the iteration and development phases were also delegated. The reported results were all analysed and verified manually by the human author.

The report was written by the human author, with suggestions from Opus 4.6. Opus 4.6 also did proofreading but all adaptations and changes were manually decided upon by the human author.

Usefulness diagrams and the implementation description in the appendix were generated by Opus 4.6