CodeScientist: End-to-End Semi-Automated Scientific Discovery with Code-based Experimentation

Peter Jansen[1,3], Oyvind Tafjord[1], Marissa Radensky[1], Pao Siangliulue[1], Tom Hope[1,4], Bhavana Dalvi Mishra[1], Bodhisattwa Prasad Majumder[1], Daniel S. Weld[1,2], Peter Clark[1]

[1]Allen Institute for Artificial Intelligence, [2]University of Washington, [3]University of Arizona, [4]Hebrew University of Jerusalem

peterj@allenai.org

Abstract

Despite the surge of interest in autonomous scientific discovery (ASD) of software artifacts (e.g., improved ML algorithms), current ASD systems face two key limitations: (1) they largely explore variants of existing codebases or similarly constrained design spaces, and (2) they produce large volumes of research artifacts (such as automatically generated papers and code) that are typically evaluated using conference-style paper review with limited evaluation of code. In this work we introduce CodeScientist, a novel ASD system that frames ideation and experiment construction as a form of genetic search jointly over combinations of research articles and codeblocks defining common actions in a domain (like prompting a language model). We use this paradigm to conduct hundreds of automated experiments on machine-generated ideas broadly in the domain of *agents and virtual environments*, with the system returning 19 discoveries, 6 of which were judged as being both at least minimally sound and incrementally novel after a multi-faceted evaluation beyond that typically conducted in prior work, including external (conference-style) review, code review, and replication attempts. Moreover, the discoveries span new tasks, agents, metrics, and data, suggesting a qualitative shift from benchmark optimization to broader discoveries.[1]

1 Introduction

Automated scientific discovery (ASD) systems have already had success in targeted domains like protein folding (AlphaFold, Jumper et al., 2021), antibiotic discovery (Stokes et al., 2020), and model optimization (Lion, Chen et al., 2023), by using custom *problem-specific* systems that search (large) hand-crafted search spaces. Recently, language models (LMs) are fueling explorations into more *problem-general* discovery systems capable of the full research pipeline of ideation, planning, (code-based) experimentation, and experiment analysis, with numerous impressive systems appearing recently, including AI Scientist (Lu et al., 2024a), AIGS (Liu et al., 2024), AgentLab (Schmidgall et al., 2025), and Data-to-Paper (Ifargan et al., 2024). Impressive as these systems are, each makes simplifications to reduce complexity, such as restricting search to variants of prewritten code, using a DSL for experiments, or working on restricted problems.

In this work we introduce CodeScientist, an ASD system built with novel innovations for ideation and experiment execution – incorporating genetic search (Hemberg et al., 2024) over combinations of literature and code – that we hypothesize will increase the diversity of the discoveries the system makes. We run our system at scale (hundreds of experiments) in the broad domain of *agents and virtual environments*, and find that of the 19 discoveries our system suggests, 6 appear to meet minimum thresholds for scientific soundness and incremental novelty after domain expert review. Moreover, the discoveries our system produces qualitatively appear diverse, and span creating new tasks, agents, metrics, data, and challenging assumptions, which builds-upon (while broadening) the scope of the impressive accomplishments of existing systems that focus on improving model performance on standardized ML benchmarks (e.g. Schmidgall et al., 2025; Li et al., 2024; Huang et al., 2024).

Orthogonally, ASD research itself is faced with significant methodological challenges that limit progress. The first, **evaluating discoveries**, is complex in that scientific discovery is (by definition) at the edge of human knowledge, and as such, tasks with gold annotated outcomes (e.g., SWE-Bench, Jimenez et al., 2023) are generally unavailable. Alternative assessments are needed, for example the way humans evaluate research – namely, rigorous (and expensive) manual review, or the use of proxy

[1]https://github.com/allenai/codescientist

Findings of the Association for Computational Linguistics: ACL 2025, pages 13370–13467

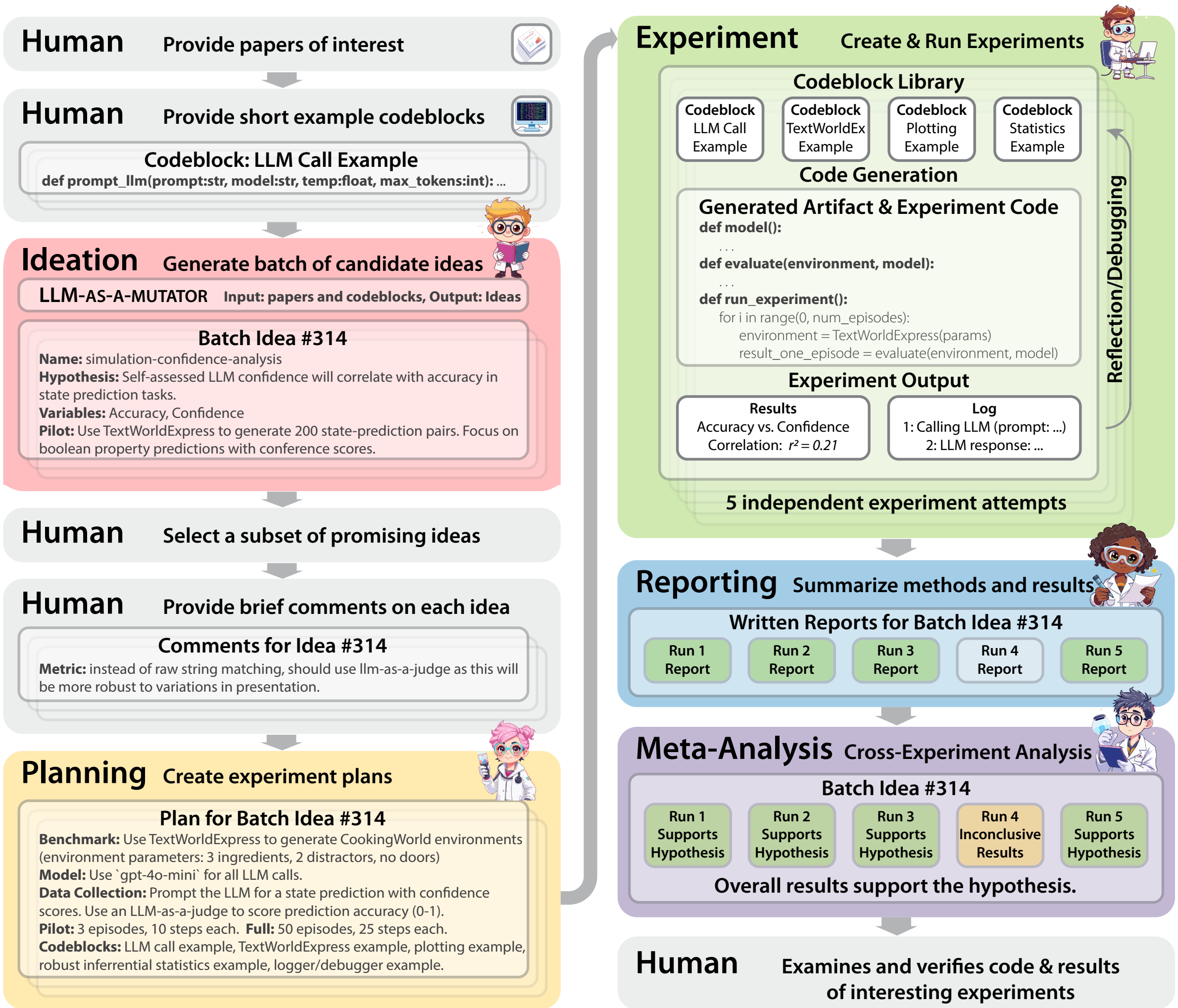

Figure 1: An overview of the core stages of the CodeScientist discovery workflow, including ideation, planning, building and executing code-based experiments, reporting results, and performing meta-analyses across experiments.

metrics (Lu et al., 2024a) that are progressively improving their agreement with human ratings (Radensky et al., 2024). The second challenge, **variability**, acknowledges that even when using low temperatures, workflows built upon language models rarely produce the same output – especially when each step autoregressively depends upon the output of the previous step. Across successive runs, the specific ideas that an ASD system generates are different, and (as we show in this work), the code it implements for those ideas (and whether it succeeds or fails) is also highly variable – making comparisons across systems (or ablations of a single system) costly, methodologically challenging, and uncommon in the literature. At the same time, with ASD still in its infancy, scientifically sound and novel discoveries are relatively rare – and (like other code-generation tasks like SWE-Bench), systems are faced with high variability and low absolute success, but (unlike SWE-Bench) without the benefit of trustworthy automatic evaluation. In spite of these methodological challenges, we show that CodeScientist is capable of generating a number of candidate discoveries through manual (domain-expert) evaluation, and that this set of discoveries qualitatively captures a series of diverse research ideas that expand the scope of accomplishments of existing systems.

The contributions of this work are:

1. CodeScientist, a novel, open-source, end-to-end system for semi-automated scientific discovery, which ideates and executes experiments based on genetic search over both literature and a library of codeblocks.

2. A demonstration in the domain of *agents and virtual environments*, where we show that CodeScientist discovers 6 incremental yet novel research results not seen previously. These results were validated by external re-

Discovery System	Ideation Methodology	Artifact Evaluated	Evaluation (Automatic)	Evaluation (Human)
AIScientist (Lu et al., 2024b)	mutate existing experiment	paper	Likert (NeurIPS)	No
AIGS (Liu et al., 2024)	task + past experiments	paper	Natural Language	Likert
AgentLab (Schmidgall et al., 2025)	human + literature	benchmark	Likert	Likert
Data-to-Paper (Ifargan et al., 2024)	table analysis	code	Rules	Code Review
MLR-Copilot (Li et al., 2024)	literature	benchmark	Likert	Likert
CodeScientist (this work)	**literature + codeblocks**	**paper+code**	**Accept/Reject Hyp.**	**Likert (paper)+ Code Review**

Table 1: A comparison of existing discovery systems with CodeScientist, in terms of their ideation methodology, which research artifact is evaluated (paper, code, or performance on a benchmark), and which automatic or manual evaluations are performed on the research results.

viewers (in a conference-style review), then further vetted by replication and code review.

3. A qualitative analysis of failure modes and research methods challenges, to help guide future research on ASD technology. This includes an examination of the benefits of incorporating human input into our workflow, versus using a completely automated system.

2 Related Work

Ideating and Executing Research: The suite of recent ASD systems primarily differ in their methods for ideation, experiment construction and execution, as well as their problem domain (e.g. chemistry, biology, AI) – with examples of systems in the AI domain provided in Table 1. Ideating on literature is a common (and naturalistic) method, but unrestricted, can produce broadly scoped ideas that are challenging to implement. Ideating on existing experiment code, essentially applying the LLM as a mutation operator (Hemberg et al., 2024) to hypothesize how code changes might (for example) increase performance on a benchmark, helps narrow ideas to those likely to be executable. Similarly, one can restrict ideation to problem-specific templates for specific tasks (Liu et al., 2024; Ifargan et al., 2024). In this work, we explore a novel genetic search ideation strategy that combines two facets: the open-endedness of literature-based ideation, with a focus on generating ideas that are implementable by our experiment builder by partially conditioning the ideation on a library of codeblocks available to the executor that implement common research functions (like calling a language model, or creating a plot).

Evaluating Discoveries: Automatic evaluation of discoveries is possible in a limited subset of domains and tasks – such as in materials science, where molecular simulators can directly evaluate the veracity of discoveries (Qi et al., 2024). Similarly, broader science-themed games can include instrumentation for measuring discoveries (Jansen et al., 2024), or the scope of discoveries can be reduced to some measurable quantity, such as increasing model performance on a benchmark task (Huang et al., 2024). Manual evaluation of research artifacts (such as code) is costly to perform, and previously limited in scope (e.g., statistical analysis code in Ifargan et al., 2024). Others have worked to develop automated proxy metrics that use an LLM-as-a-Judge paradigm to evaluate research papers on Likert scales (Likert, 1932) similar to conference review (Lu et al., 2024a; Schmidgall et al., 2025; Li et al., 2024), though due to the difficulty of this task, these currently may have limited agreement with human judgements (Radensky et al., 2024), may be fooled by superficial niceities (fluency, layout, etc.), and (as in the case of AI Scientist), may reject all the discoveries.[2] As we show in Section 4, faithfulness is a critical factor to automated evaluation – what our system says it has discovered in its papers can strongly differ from what it actually implements in code, and we argue that conference-style Likert assessments of papers (without code) may not fully assess the faithfulness of science-as-code discoveries.

3 System Overview

An overview of the CodeScientist system is shown in Figure 1, with the 5 major steps of the workflow (*ideation*, *planning*, *experiment building*, *reporting*, and *meta-analysis*) described below. Except where

[2] AI Scientist recently announced that it submitted 3 AI-generated papers from an unreleased (v2) system to a workshop, one of which was accepted: https://sakana.ai/ai-scientist-first-publication-jp/; Similarly intology.ai recently reported two AI-generated papers accepted for workshops (Intology AI, 2025).

described otherwise, the steps of the workflow are implemented as prompts to a language model, with example prompts and additional implementation details provided in APPENDIX F.

3.1 Paper Corpus and Codeblocks

CODESCIENTIST requires two forms of pre-generated input: (a) a human-curated list of papers to ideate from in the users domain(s) of interest, and (b) a set of relevant codeblocks that demonstrate how to perform common tasks. Example codeblocks include *how to call an LLM*, *how to implement a ReAct agent*, and *how to load specific benchmarks*. More on the specific agent-centered corpus of papers and codeblocks used in this work is provided in Section 4.

3.2 Ideation

The purpose of the ideator is to generate a large set of candidate research ideas (conditioned on recent research articles) that could be explored by CODESCIENTIST. Pilot studies showed that nearly all of the ideas generated by our early ideator were either too complex or too open-ended to be implemented by our automated experiment building system, such as requiring the ability to download and modify arbitrary models and benchmarks mentioned in the input papers. To generate ideas that are both conditioned on the literature and on code components that have a high chance of successfully executing in the execution sandbox, the ideator prompt includes both (a) two randomly chosen papers, from a corpus of papers provided by a human scientist, and (b) summaries of codeblocks that form a vetted library of common research-related functions. More formally, the ideator takes a human-generated corpus of papers ($papers$) and a library of codeblocks ($codeblocks$) as input, producing a set of candidate ideas ($ideas$) as output:

$$ideas = \text{Ideator}(papers, codeblocks) \quad (1)$$

Each idea ($i \in ideas$) is structured with slots for the *hypothesis*, *dependent/independent variables*, *evaluation metrics*, *baselines*, *pilot experiment design*, and a list of *major sections of code and other resources* anticipated to be required to successfully implement the idea, with slot values populated in natural language. To emphasize creating diverse combinations of ideas conditioned on ideas in the literature while using genetic-style search (Hemberg et al., 2024), the prompt includes sample types of ideation to serve as genetic operators, including cross-over (i.e. *combining ideas*), and mutation (i.e. *extending ideas*, *challenging assumptions*, *filling in gaps*, and so forth). After generating a large pool of candidate ideas, as a cost-saving and efficiency measure, a human then manually selects a subset of these ideas, $ideas_f \subset ideas$, that appear most interesting to them. For each idea $i \in ideas_f$, the domain expert can also provide a brief (2-3 sentence) set of human comments h, such as suggesting alternate metrics or benchmarks to use, that increase the utility and tractability of the idea. An example idea, human comment, and plan is provided in APPENDIX E.[3]

3.3 Planning

The planning step converts the high-level idea generated by the ideator into a more detailed, practical, and operational experiment plan for the experiment builder. Where the ideator generates high-level ideas such as *"examine whether a ReAct agent augmented with a causal memory increases performance on benchmark X"*, the planning step generates a specific plan for implementing the *artifact* (i.e. the augmented REACT agent) as well as the *experiment* to test its properties (such as which base model to use, hyperparameters, and other experimental details). More formally, the Planner takes as input an idea i, expert comments on that idea h, and the codeblock library C as input, producing a plan p and anticipated list of codeblocks $c \subset C$ required to implement that plan as output:

$$p, c = \text{Planner}(i, h, C) \quad (2)$$

This experiment plan and list of codeblocks serve as input to the experiment construction system.

3.4 Experiment Construction and Execution

The experiment builder is tasked with generating the code for the *artifact* and *experiment* through an iterative series of *generate-execute-reflect* debugging steps. The builder takes an experiment plan p and list of codeblocks c as input, and produces a set of generated code g, experimental results r, and output logs l as output:

$$g, r, l = \text{Builder}(p, c) \quad (3)$$

[3]The full set of human comments is provided in APPENDIX D, and an ablation in the discussion suggests removing expert comments reduces the number of discoveries judged as minimally sound and having at least incremental novelty by approximately one third.

Rating	Automated Result Summary
supports	Discovered action templates significantly improved agent performance in CookingWorld, outperforming both baseline and manual templates.
inconcl.	Decomposition history slightly improved agent performance (0.183 vs 0.117) but results weren't statistically significant.
reject	Knowledge graph-based mode switching showed no clear advantage over baseline strategies in ScienceWorld tasks.

Table 2: Example result summaries and corresponding ratings for whether they *support, reject*, or are *inconclusive* towards the respective experiment hypothesis. Each of the 3 examples comes from a different idea.

The experiment builder includes three tightly-coupled components:

Initial code generation: Synthesizes PYTHON code to implement the artifact and experiment based on the plan and selected codeblocks.

Instrumented execution sandbox: The generated code is executed within an instrumented sandbox. This sandbox captures the full output of the code – including logging and standard output/error streams, and intercepts API calls (e.g. to OPENAI, ANTHROPIC, or TOGETHER.AI models) via a proxy that tracks usage statistics and enforces cost limits.

Reflection and debugging: At each debug iteration, the model is asked to reflect on the code and output (logs, streams, and API usage statistics), and determine if the experiment has completed successfully and faithfully. If successful, the experiment building step concludes and progresses to reporting. If unsuccessful, the reflection step modifies the code, and the *execute-reflect* cycle continues until the experiment is marked as complete, or a hard limit (time, cost, or number of debug iterations) is reached.

For pragmatic reasons (cost, runtime, prompt length), the experiment builder first attempts to build a short pilot experiment with inexpensive and fast debug cycles.When the experiment builder determines the pilot experiment has succesfully completed, it automatically scales to running (and, if necessary, further debugging) the full experiment.

3.5 Reporting

Successfully completed experiments enter an automated reporting step that takes the experiment plan (p), code (g), results (r), and logs (l) as input and produces both a written LaTeX report w, and a short summary report s as output:

$$w, s = \text{Reporter}(p, g, r, l) \tag{4}$$

Because the system can have a high throughput and generate experimental results faster than a human could read the detailed reports (w), the short high-level summaries of results (s) help a user determine if reading the full report is warranted by highlighting the main results. This includes explicitly categorizing whether the hypothesis of the experiment was *confirmed* or *rejected* by the experimental results, or if the results were *inconclusive*. Example summaries and ratings are shown in Table 2.

3.6 Meta-Analysis

Pragmatically, even when provided with the same plan, codeblocks, and a low generation temperature, the experiment builder frequently produces different implementations across successive runs due to the inherent variability introduced both by the language models themselves, as well as by autoregressively conditioning the code they generate on the output of the previous debug cycle. At the same time, language models still struggle with many complex scientific code generation tasks, and some experiments may fail to create a successful implementation. To reduce this variability, we include a meta-analysis step where for each idea and plan, the experiment builder is independently run N times, producing N different experimental results. The meta-analysis then examines the consistency of the results across successive experimental implementations, generating a meta-analysis report m:

$$m_i = \text{MetaAnalysis}(s_1, s_2, s_3, ..., s_N) \tag{5}$$

where s_n represents a single result summary from running a given plan p through the experiment builder and reporting process N times.

4 Discovery Experiments

Candidate discoveries made by CODESCIENTIST are described in Section 4.1, with challenges and failure modes described in Section 5. A description of the experiment setup is provided below.[4]

Paper and Code Example Corpus: We assembled a corpus of 57 recent papers broadly in the area of agent architectures and virtual environments. Paired with this, we assembled 10 example code snippets[5] for performing basic tasks in this domain, including calling language models (OpenAI et al.,

[4]Unless otherwise stated, we use CLAUDE-SONNET-3.5-1022 as our base model for CODESCIENTIST. We ask the planner to prefer to experiment on GPT-4O-MINI, due to its high speed and low cost.

[5]Code snippets were assembled from a combination of existing library examples and documentation, LLM-generation, and/or manual authoring.

Configuration	
Number of ideas evaluated	50
Experiment Builder attempts per idea	5
Total experiment attempts	250
Enforced Limits (per experiment)	
Maximum Debug Iterations	25
Total cost limit	\$10
LLM cost limit (per debug iteration)	\$1
Execution time limit (per debug iteration)	90 min.
Hard time limit (all debug iterations)	6 hours
Average Usage (per experiment; N=250)	
Average debug iterations	15.8
Average cost	\$4.23
Average runtime	131 min.
Average generated code length (lines)	506
Average generated code length (tokens)	4521

Table 3: Summary statistics for discovery experiments.

2024), building a ReAct agent (Yao et al., 2023), plotting (Hunter, 2007), robust inferential statistics for comparing models (Efron, 1992), using common knowledge graphs (Speer et al., 2017; Miller, 1995), and several benchmark environments including TEXTWORLDEXPRESS (Jansen and Cote, 2023), a high-performance simulator that reimplements common benchmarks. These code examples serve not only as vetted code examples of common artifacts described in the paper corpus, but also as demonstrations of the nuances required to implement code within the sandbox environment.

Idea Selection: We used CODESCIENTIST to generate approximately 2000 candidate experiment ideas ideated from 200 randomly selected combinations of papers. As a cost saving measure, a stratified sample was presented to a domain expert, until they had manually selected 50 ideas that appeared both viable and sufficiently different from one another. Each idea was provided with brief comments by the domain expert (included in APPENDIX D, to show the minor corrections these entail), then plans for each idea were generated.

Experiment Builder: For each of the 50 ideas, we called the experiment builder 5 times (i.e. each idea was given 5 attempts to generate functioning experiment code and generate results), for a total of 250 experiment runs. Cost, runtime limits, and actual usage statistics are provided in Table 3.

Reporting and Meta-Analysis: The reporting stage produced written reports for each experiment (intended for the user, and provided in APPENDIX H), as well as short summaries of results. Short summaries of each of the 5 experiments for a given idea were used for meta-analysis, which described whether the results across experiments generally support or reject the hypothesis. Additional details of the meta-analysis procedure are provided in APPENDIX C.

4.1 Candidate Discoveries

CODESCIENTIST flagged 19 of 50 ideas for human inspection where at least one of the five experiment runs produced "interesting" results,[6] with those discoveries provided in Table 4. We performed two separate evaluations on these candidate discoveries: external, and internal.

External (conference-style) Review: A conference style review. We recruited 3 external reviewers that are practicing research scientists in natural language processing and who have previously published in the domain (agents and virtual environments). Reviewers were provided with CODESCIENTIST generated papers, and asked to rate them on *soundness* and *novelty*, with the rubric provided in APPENDIX B. Each domain expert rating was then converted to a binary score. Ratings of *unsound* or *not novel* experiments were considered failures, and converted to zeros. Ratings of *clearly sound*, *likely sound*, or *minor concerns (not altering overall conclusions)* were considered meeting the minimum threshold for soundness, and converted to scores of one. Similarly, ratings of *incremental novelty* or above were considered meeting the minimum threshold for novelty, and converted to one. The average of these binarized ratings across the three external reviewers is provided in Table 4. If the majority of reviewers rated a discovery as meeting minimum soundness and novelty thresholds, we considered it as having passed external review.

Internal Review: A domain expert (one of the authors) provided an in-depth review of the code and experiment logs, and attempted to replicate the results with a larger number of samples. This reviewer essentially functioned as a "veto", able to reject discoveries that appeared genuine from the paper and external review, but did not pass detailed examination.

Candidate discoveries: Of the 19 candidate discoveries flagged by CODESCIENTIST, 13 (68%) were rated as meeting minimum soundness and novelty criteria by at least 2 of the 3 external reviewers.

[6] Results were flagged for humans using a heuristic prompt found in APPENDIX F

#	Human Reviewers Min. Sound	Some Novelty	Description of Discovery
Candidate discoveries identified by external reviewers, and supported by internal review			
1	1.0	1.0	**State Prediction Confidence:** In a state prediction task, an LLM's self-assessed confidence in its predictions have a low corelation with the accuracy of those predictions. (The state prediction data was automatically crawled from one of the benchmarks) *(Though the correlation varies across experiments, the value consistently appears low.)*
2	1.0	1.0	**Accuracy vs Representational Expressivity:** In a state prediction task, an LLM performs better at predicting simpler representations (e.g. boolean values) versus states including text. (The state prediction data was automatically crawled from one of the benchmarks) *(Significant implementation and evaluation differences across experiments, but generally support the idea that predicting simpler representations is easier.)*
3	1.0	1.0	**Multi-Stage Environment Generation:** When creating novel benchmark environments using code-generation, generating the environments in multiple stages increases environment fidelity. *(A small change on LLM-for-environment-generation tasks, implementing specific aspects in each step, rather than generating as a whole and reflecting. For evaluation, creates a simple proxy metric that seems well-motivated as this type of evaluation is an open problem in the literature, and even llm-as-a-judge paradigms have issues with this task, while being vastly more expensive).*
4	1.0	1.0	**Combinatorial Optimization:** A language model performs poorly at a combinatorial optimization problem (selecting values from a set that are closest to adding to a specified value X), grounded in substituting resistor values in electronics. *(Consistent result across experiments, and tested to within different tolerances, e.g. 1%, 5%)*
5	1.0	0.66	**Action Prediction:** An LLM's ability to predict whether actions will be successful in a virtual environment is generally low, marginally above a random baseline. *(Appears true, with the following qualifications: (1) the LLM was given only the current observation, and no history, to judge from, and (2) an LLM-as-a-judge was used to help collect the gold dataset, and has imperfect labels.)*
6	0.66	0.66	**Graph Agent for Discovery:** A ReAct agent augmented with a graph-based memory outperforms a ReAct baseline on a highly complex environment (DiscoveryWorld). *(Appears true. Graphs appear relatively simple, forming a form of OBJECT-PROPERTY memory, rather than complex nested relationships.)*
Results rejected by either external reviewers, internal review, or both			
7	1.0	1.0	**Social Graphs:** For a task in managing social relationships, an agent that uses a graph to keep track of those relationships does not outperform perform a simpler non-graph baseline. *(The experiment-generated benchmark is overly simple, contains few samples, and the llm-as-a-judge metric is not validated)*
8	1.0	1.0	**Planning Agent:** A custom planning agent outperforms a ReAct agent on the CookingWorld benchmark. *(Effect fails to replicate when number of samples increased).*
9	1.0	1.0	**Spatial Agent:** A ReAct agent that maintains an explicit graph of interconnected locations outperforms a baseline ReAct agent. *(Effect fails to replicate when number of samples increased).*
10	1.0	1.0	**Action History:** A ReAct agent that keeps a history of recent actions (as well as whether those actions increased the task score) outperforms a ReAct baseline. *(Effect fails to replicate when samples increased).*
11	1.0	0.66	**Template Agent:** Creates an agent that applies templates of action sequences learned in the training set. *(A non-LLM agent that is partly hard-coded, and partly reinventing a Markov model).*
12	0.66	1.0	**Graph Verification Agent:** An agent that builds and explicitly verifies its graph before using it outperforms a baseline agent without this verification step. *(On examining the code, the experiment is incorrect because neither baseline nor experimental agents ever actually use the graph it builds, they just randomly pick actions. An interesting secondary result – that an LLM used to extract triples from an environment observation only verifies ≈80% of those triples as correct in a secondary verification step – does appear to be supported.)*
13	0.66	0.66	**Affordance Agent:** An LLM agent that predicts object affordances outperforms a random baseline on ScienceWorld. *(Baseline is too simple)*
14	0.33	0.66	**WordNet:** A hardcoded WordNet agent outperforms a random baseline. *(Baseline is too simple)*
15	0.33	0.66	**Metaphor Graphs:** Examines generating graphs that describe the metaphorical relationships between objects in virtual environments. *(Unclear motivation and utility)*
16	0.0	1.0	**Goal Tracking:** A ReAct agent augmented with goal tracking outperforms a random baseline. *(Baseline is too simple)*
17	0.0	1.0	**Container Agent:** An agent built specifically to handle objects inside containers (a challenge for some agents) outperforms a random baseline. *(Baseline is too simple)*
18	0.0	0.66	**Template-based Environment Creation:** Parametrically generating new environment benchmarks using templates to define the environment is just as good as manually building the environments. *(Both the template and "manual" (should be human-generated?) environments were hardcoded in the experiment code – so this was a foregone conclusion).*
19	0.0	0.66	**Graph Metric:** A custom-developed metric for determining similarity between text and graph-based representations of environments outperforms Jaccard similarity. *(Report lacks details to evaluate.)*

Table 4: Descriptions of the 19 discoveries CodeScientist automatically identifies from 250 experiments over 50 ideas. *Human Reviewers* refers to average ratings from 3 external reviewers. The 6 discoveries under "candidate discoveries" were rated as meeting minimum soundness and and incremental novelty criteria by external reviewers (examining papers), and similarly by the internal reviewer (examining both code and papers). The 11 discoveries under "rejected" received low ratings from the external reviewers, or received high ratings from external reviewers but were rejected by the internal reviewer either after examining the code and identifying issues with soundness, or failing to replicate results after rerunning experiments with a higher numbers of samples. Comments from the internal reviewer are provided in *italics*.

When examining the code, experiment logs, and performing replication attempts, the internal reviewer rejected 7 of these 13 discoveries, resulting in a total of 6 discoveries (32%) that passed both external and internal review.

The discoveries passing these tests take a variety of forms. While some take the form of improving model performance on benchmarks, most involve creating new tasks, benchmarks, metrics, methods, or questioning assumptions. The discoveries include determining that an LLM's self-assessed confidence in its prediction accuracy has a low correlation with its actual accuracy in state-prediction tasks (#1) – a result ideated from a paper on assessing only accuracy in state prediction for virtual environments (Wang et al., 2024), and whose benchmark was unavailable in the sandbox, causing the experiment builder to crawl one of the environments that was accessible to it with a random agent to create *its own benchmark* for the experiment. Similarly (and also on state-prediction), a different discovery suggested that an LLM's performance on state prediction tasks varies with the complexity of the representation it has to predict (#2). Another discovery found that language models are particularly poor at assessing whether an action will be successful in an environment given the previous environmental observation (#5). While language models' arithmetic performance is well studied (e.g. Yuan et al., 2023), a discovery suggested they are poor at solving a specific combinatorial optimization problem involving addition (#4). Reports and code for these discoveries are provided in Appendix H, with assessments of (incremental) novelty in Appendix G.

Rejected discoveries: 13 of 19 discoveries were rejected after human evaluation. These include 6 reported discoveries where the baselines or models themselves were overly simplistic, and 3 reported discoveries where the reported effect disappeared after rerunning the experiment with a larger number of samples. 2 reported discoveries had major implementation errors discovered by the domain expert, and were rejected. One candidate discovery was rejected by external reviewers for having limitd details from which to evaluate its soundness.

5 Discussion

We outline challenges and common failure cases below, to highlight where targeted efforts might improve discovery quality.

Experiment Builder Outcomes	%
Experiment completed successfully	41%
Debug iteration limit reached	32%
Hard experiment time limit reached	18%
Unrecoverable code generation issue *(i.e. code too long for output)*	9%
Hard cost limit reached	0%
Number of samples (experiments)	250

Table 5: Summary statistics of the *experiment builder*.

Idea Diversity: While the candidate discoveries demonstrate that the ideator can generate diverse ideas that span designing new agents, tasks, metrics, methods, and benchmarks, most of the ideas that are generated are highly similar, or mechanical variations of one another (e.g. *apply method X to benchmark Y*). Si et al. (2024) observed that LLM-ideators quickly saturate the number of unique ideas they generate. We found that using their method of uniqueness filtering (using cosine similarity over embeddings) still generated many duplicates, and had to use manual filtering to select a diverse, fairly non-overlapping subset to explore.

Experiment Builder Failures: Currently, 59% of the experiments the planner designs are not able to be successfully implemented by the experiment builder. The distribution of executor errors is shown in Table 5. The majority of experiments ending in failure face challenges in debugging that are never resolved, hitting either the maximum number of debug iterations (32%) or experiment time limits (18%). Infrequently, the experiment code is too long to fully generate (9%), requiring a model that can generate more than 8k output tokens.

Unfaithful Experiments: The system occasionally reports a result that (upon closer examination of the code) is untrue. For example, the system reported that a graph-based agent with a verification mechanism outperformed a baseline agent *(Result #12 in Table 4)*. Human inspection of the code revealed that while the agent built the graph, it never used it, instead picking actions randomly. These types of errors are difficult and laborious to detect, requiring a great deal of human effort. Methods to automate this will help reduce false positives/negatives.

Adherence to best research practices: Many generated experiments do not adhere to best practices in research methods, even when prompted to do so. Experiments may evaluate on the training set, or make errors in calculating or interpreting statistics. They may similarly choose weak baselines to compare experimental models against (e.g. comparing modified ReAct agents to random baselines rather than ReAct baselines, as in Table 4), erroneously producing statistically significant results.[7]

Reducing Human Effort: Our system currently uses human involvement at 5 steps – a pragmatic requirement given that when run in fully-automated mode, an early pilot of CodeScientist generated results much less efficiently: ideas were frequently duplicated, experiments failed more often during debugging, and results had methods challenges and were less convincing. Currently human involvement is included in selecting papers to ideate from, generating codeblocks, selecting interesting ideas to run, providing brief expert comments on each ideas, and performing a manual validation of the results. Towards the goal of automation, a list of representative papers could be provided by a paper selection agent in response to a broad area of interest provided by a user. Many of the codeblocks used by the system were already produced by a language model, but manually corrected and tweaked by a human for running in the sandbox. The rapid progress in ideation models (e.g. Wang et al., 2023a) and evaluation (Radensky et al., 2024) is likely to produce systems that continue to increase idea diversity and utility in the near-term, reducing the need for a human to filter these ideas. Our use of brief domain-expert comments on ideas before generating plans helps the system shore up obvious research methods issues (like using a better metric) before running costly experiments, and as models increase their knowledge of domain-specific research methods, this reliance will decrease.

Fully-automated Mode: While we present CodeScientist as requiring human input, it is possible to run CodeScientist in a fully-automated mode without such input. Here we present two ablations that remove human input, with the caveats that (a) the variance of the system (even when provided with identical experiment plans) is high, (b) the overall candidate discovery rate is low, and (c) large-scale runs are prohibitively expensive – which, taken together, makes performing ablations with enough statistical power to make inferences currently impractical. That being said, in pilot experiments, running CodeScientist on 100 ideas generated and executed autonomously (without human input) appeared to generate 2 candidate discoveries that passed an internal review, or a 2% success rate (and, our motivation for pre-filtering ideas to those most likely to produce discoveries). Similarly, rerunning the benchmark of 50 ideas described in Section 4 *without* expert human comments appears to produce 4 of the same 6 candidate discoveries as were found in Table 4 – nominally reducing the number of candidate discoveries by approximately a third. It is important to note that this is a small sample, and difficult to ascertain whether this reduction is due to the lack of expert comments, or the natural variance in the system.

6 Conclusion

We present CodeScientist, an end-to-end semi-automated discovery system with a novel ideator that performs genetic search jointly on combinations of literature and codeblocks, before building, running, and analyzing these software artifacts in experiments. Ideating from literature and common codeblocks in the domain of *agents and environments*, CodeScientist identified 19 potential discoveries, 6 of which were judged as meeting minimum thresholds for scientific soundness and incremental novelty after both external conference-style review, and an internal code review. A qualitative review of the discoveries suggests that they broaden the scope of ideas explored by other systems, with candidate discoveries spanning novel tasks, agents, metrics, and data. We provide an analysis of common failure modes, and release our system as open-source to facilitate future research in automated scientific discovery.

7 Limitations

We highlight the following limitations both in this work, as well as in current automated discovery systems more broadly.

Cost vs accuracy trade-off: The automated experiments in this work are designed to be fast, inexpensive estimates of an experiment's results, that allow rapid iteration. The average experiment in this work costs ≈$4, and takes approximately 2 hours to complete. While these experiments save on resources, due to their low number of samples, they both produce false positives, and are likely unable to detect all but the largest effects (producing false negatives). This is not a technical limitation

[7] In our detailed code analysis, we overlooked research methods issues that were unlikely to invalidate results – for example, evaluating on the training set with *zero-shot* methods that do not fit any parameters using the training set – though this is unconventional, and limits the ability to compare these results to those reported in other papers.

of this work as (in principle) it can be remedied by scaling the computation budget – though developing strategies for intelligency investing in the experiments most likely to have utility could reduce budget requirements.

Validating Candidate Discoveries: The common mode of disseminating research artifacts in NLP/AI is through peer-reviewed scientific articles. This peer review typically examines the article in detail, but reviewers are frequently instructed that they are not required to provide a review of supplementary material including code[8] – likely in part because code review is extremely labor intensive, but also because a degree of skill, domain-specific training, and a good-faith effort are assumed on behalf of the authors. The opposite appears true of language-model-generated code, and in this work we show that more than half of the potential discoveries were rejected by an internal code review by a domain expert (one of the authors) for having serious issues. We observed – both in pilot experiments, as well as in evaluating the 19 potential discoveries – that the LLM-generated code may unfaithfully represent the desired process or mechanism requested in the experiment plan, and instead actually perform a much simpler procedure (like randomly generating actions to take in a virtual environment). These are particularly challenging to find because, often, much of what the LLM claims is happening is at least partially implemented, and the offending code may be only one (or a few) lines in a long program. As such, while we have made a good-faith effort to validate the discoveries and code proposed by CodeScientist– including examining the code and rerunning each discovery at a greater scale, which is arguably a more effortful review than is done for most peer-reviewed articles – given the sometimes adversarial nature of the task, it is possible we may not have discovered some inaccurately reported mechanisms. As such, we frame the 6 discoveries made by the system (and having passed external and internal review) as "candidate discoveries", to emphasize their preliminary nature, and the challenges of performing review on long LLM-generated code. Developing automated mechanisms to speed this review is likely of paramount importance to scaling automated scientific discovery systems that use code-based experimentation.

[8]https://aclrollingreview.org/reviewerguidelines

Incremental vs Transformational Discoveries: The 6 expert-validated discoveries produced by CodeScientist would likely be categorized by most as normal incremental science rather than transformational discoveries – and the ratings by the 3 external reviewers suggest that each candidate CodeScientist discovery is (at best) incrementally novel. For example, while the first result (#1) in Table 4 finds that an LLM's self-assessed confidence in its predictions has a low correlation with its actual performance on state prediction tasks focusing on using LLMs as world models, similar effects have been shown on other tasks, and testing the existence of this phenomenon on state prediction in virtual environments is an incremental discovery. The discoveries produced by CodeScientist are incremental rather than transformational discoveries, and we have no data to support whether generating more impactful discoveries is a problem of scale (i.e. running either more ideas, or higher risk/higher gain ideas), or a problem of kind (i.e. whether ideation and execution in the manner that we have described here is capable or incapable of generating high-impact discoveries).

Ideator Recall: Studying the "recall" of an ideator (in terms of the proportion of ideas that it generates that ultimately provide discoveries that are judged to be at least minimally sound and incrementally novel) is an open area of research. In pilot experiments we generated new ideas at runtime, many of which had issues (such as being near duplicates, using incorrect metrics, or being very challenging to implement). As a pragmatic cost saving measure, we explicitly filter down a large set of ideas to manually select the first 50 ideas that appeared to meet the bar of being potentially implementable, relatively different from one another, and that did not appear to have obvious research methods problems. As such, due to this cost saving measure, we are unable to make claims about what proportion of ideas that the ideator generates ultimately lead to human-verified discoveries.

References

Xiangning Chen, Chen Liang, Da Huang, Esteban Real, Kaiyuan Wang, Yao Liu, Hieu Pham, Xuanyi Dong, Thang Luong, Cho-Jui Hsieh, Yifeng Lu, and Quoc V. Le. 2023. Symbolic discovery of optimization algorithms. *ArXiv*, abs/2302.06675.

Bradley Efron. 1992. Bootstrap methods: another look at the jackknife. In *Breakthroughs in statis-*

tics: Methodology and distribution, pages 569–593. Springer.

Jiahui Geng, Fengyu Cai, Yuxia Wang, Heinz Koeppl, Preslav Nakov, and Iryna Gurevych. 2024. A survey of confidence estimation and calibration in large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 6577–6595, Mexico City, Mexico. Association for Computational Linguistics.

Yuling Gu, Bhavana Dalvi Mishra, and Peter Clark. 2023. Do language models have coherent mental models of everyday things? In *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1892–1913, Toronto, Canada. Association for Computational Linguistics.

Erik Hemberg, Stephen Moskal, and Una-May O'Reilly. 2024. Evolving code with a large language model. *Genet. Program. Evolvable Mach.*, 25:21.

Qian Huang, Jian Vora, Percy Liang, and Jure Leskovec. 2024. Mlagentbench: Evaluating language agents on machine learning experimentation. In *Forty-first International Conference on Machine Learning*.

John D Hunter. 2007. Matplotlib: A 2d graphics environment. *Computing in science & engineering*, 9(03):90–95.

Tal Ifargan, Lukas Hafner, Maor Kern, Ori Alcalay, and Roy Kishony. 2024. Autonomous llm-driven research from data to human-verifiable research papers. *ArXiv*, abs/2404.17605.

Intology AI. 2025. Zochi technical report. Technical report, Intology AI.

Peter Jansen. 2022. A systematic survey of text worlds as embodied natural language environments. In *Proceedings of the 3rd Wordplay: When Language Meets Games Workshop (Wordplay 2022)*, pages 1–15, Seattle, United States. Association for Computational Linguistics.

Peter Jansen and Marc-alexandre Cote. 2023. TextWorldExpress: Simulating text games at one million steps per second. In *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics: System Demonstrations*, pages 169–177, Dubrovnik, Croatia. Association for Computational Linguistics.

Peter Jansen, Marc-Alexandre Côté, Tushar Khot, Erin Bransom, Bhavana Dalvi Mishra, Bodhisattwa Prasad Majumder, Oyvind Tafjord, and Peter Clark. 2024. Discoveryworld: A virtual environment for developing and evaluating automated scientific discovery agents. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

Carlos E Jimenez, John Yang, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, and Karthik R Narasimhan. 2023. Swe-bench: Can language models resolve real-world github issues? In *The Twelfth International Conference on Learning Representations*.

John M. Jumper, Richard Evans, Alexander Pritzel, Tim Green, Michael Figurnov, Olaf Ronneberger, Kathryn Tunyasuvunakool, Russ Bates, Augustin Žídek, Anna Potapenko, Alex Bridgland, Clemens Meyer, Simon A A Kohl, Andy Ballard, Andrew Cowie, Bernardino Romera-Paredes, Stanislav Nikolov, Rishub Jain, Jonas Adler, Trevor Back, Stig Petersen, David Reiman, Ellen Clancy, Michal Zielinski, Martin Steinegger, Michalina Pacholska, Tamas Berghammer, Sebastian Bodenstein, David Silver, Oriol Vinyals, Andrew W. Senior, Koray Kavukcuoglu, Pushmeet Kohli, and Demis Hassabis. 2021. Highly accurate protein structure prediction with alphafold. *Nature*, 596:583 – 589.

Ruochen Li, Teerth Patel, Qingyun Wang, Qingyun Wang, and Xinya Du. 2024. Mlr-copilot: Autonomous machine learning research based on large language models agents. *ArXiv*, abs/2408.14033.

Rensis Likert. 1932. A technique for the measurement of attitudes. *Archives of Psychology*.

Zijun Liu, Kai Liu, Yiqi Zhu, Xuanyu Lei, Zonghan Yang, Zhenhe Zhang, Peng Li, and Yang Liu. 2024. Aigs: Generating science from ai-powered automated falsification. *ArXiv*, abs/2411.11910.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob N. Foerster, Jeff Clune, and David Ha. 2024a. The ai scientist: Towards fully automated open-ended scientific discovery. *ArXiv*, abs/2408.06292.

Chris Lu, Cong Lu, Robert Tjarko Lange, Jakob N. Foerster, Jeff Clune, and David Ha. 2024b. The ai scientist: Towards fully automated open-ended scientific discovery. *ArXiv*, abs/2408.06292.

Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, et al. 2023. Self-refine: Iterative refinement with self-feedback. In *Thirty-seventh Conference on Neural Information Processing Systems*.

George A Miller. 1995. Wordnet: a lexical database for english. *Communications of the ACM*, 38(11):39–41.

OpenAI, Josh Achiam, Steven Adler, Sandhini Agarwal, Lama Ahmad, Ilge Akkaya, Florencia Leoni Aleman, Diogo Almeida, Janko Altenschmidt, Sam Altman, Shyamal Anadkat, Red Avila, Igor Babuschkin, Suchir Balaji, Valerie Balcom, Paul Baltescu, Haiming Bao, Mohammad Bavarian, Jeff Belgum, Irwan Bello, Jake Berdine, Gabriel Bernadett-Shapiro, Christopher Berner, Lenny Bogdonoff, Oleg Boiko, Madelaine Boyd, Anna-Luisa Brakman, Greg Brockman, Tim Brooks, Miles Brundage, Kevin Button, Trevor Cai, Rosie Campbell, Andrew Cann, Brittany Carey, Chelsea Carlson, Rory Carmichael, Brooke

Chan, Che Chang, Fotis Chantzis, Derek Chen, Sully Chen, Ruby Chen, Jason Chen, Mark Chen, Ben Chess, Chester Cho, Casey Chu, Hyung Won Chung, Dave Cummings, Jeremiah Currier, Yunxing Dai, Cory Decareaux, Thomas Degry, Noah Deutsch, Damien Deville, Arka Dhar, David Dohan, Steve Dowling, Sheila Dunning, Adrien Ecoffet, Atty Eleti, Tyna Eloundou, David Farhi, Liam Fedus, Niko Felix, Simón Posada Fishman, Juston Forte, Isabella Fulford, Leo Gao, Elie Georges, Christian Gibson, Vik Goel, Tarun Gogineni, Gabriel Goh, Rapha Gontijo-Lopes, Jonathan Gordon, Morgan Grafstein, Scott Gray, Ryan Greene, Joshua Gross, Shixiang Shane Gu, Yufei Guo, Chris Hallacy, Jesse Han, Jeff Harris, Yuchen He, Mike Heaton, Johannes Heidecke, Chris Hesse, Alan Hickey, Wade Hickey, Peter Hoeschele, Brandon Houghton, Kenny Hsu, Shengli Hu, Xin Hu, Joost Huizinga, Shantanu Jain, Shawn Jain, Joanne Jang, Angela Jiang, Roger Jiang, Haozhun Jin, Denny Jin, Shino Jomoto, Billie Jonn, Heewoo Jun, Tomer Kaftan, Łukasz Kaiser, Ali Kamali, Ingmar Kanitscheider, Nitish Shirish Keskar, Tabarak Khan, Logan Kilpatrick, Jong Wook Kim, Christina Kim, Yongjik Kim, Jan Hendrik Kirchner, Jamie Kiros, Matt Knight, Daniel Kokotajlo, Łukasz Kondraciuk, Andrew Kondrich, Aris Konstantinidis, Kyle Kosic, Gretchen Krueger, Vishal Kuo, Michael Lampe, Ikai Lan, Teddy Lee, Jan Leike, Jade Leung, Daniel Levy, Chak Ming Li, Rachel Lim, Molly Lin, Stephanie Lin, Mateusz Litwin, Theresa Lopez, Ryan Lowe, Patricia Lue, Anna Makanju, Kim Malfacini, Sam Manning, Todor Markov, Yaniv Markovski, Bianca Martin, Katie Mayer, Andrew Mayne, Bob McGrew, Scott Mayer McKinney, Christine McLeavey, Paul McMillan, Jake McNeil, David Medina, Aalok Mehta, Jacob Menick, Luke Metz, Andrey Mishchenko, Pamela Mishkin, Vinnie Monaco, Evan Morikawa, Daniel Mossing, Tong Mu, Mira Murati, Oleg Murk, David Mély, Ashvin Nair, Reiichiro Nakano, Rajeev Nayak, Arvind Neelakantan, Richard Ngo, Hyeonwoo Noh, Long Ouyang, Cullen O'Keefe, Jakub Pachocki, Alex Paino, Joe Palermo, Ashley Pantuliano, Giambattista Parascandolo, Joel Parish, Emy Parparita, Alex Passos, Mikhail Pavlov, Andrew Peng, Adam Perelman, Filipe de Avila Belbute Peres, Michael Petrov, Henrique Ponde de Oliveira Pinto, Michael, Pokorny, Michelle Pokrass, Vitchyr H. Pong, Tolly Powell, Alethea Power, Boris Power, Elizabeth Proehl, Raul Puri, Alec Radford, Jack Rae, Aditya Ramesh, Cameron Raymond, Francis Real, Kendra Rimbach, Carl Ross, Bob Rotsted, Henri Roussez, Nick Ryder, Mario Saltarelli, Ted Sanders, Shibani Santurkar, Girish Sastry, Heather Schmidt, David Schnurr, John Schulman, Daniel Selsam, Kyla Sheppard, Toki Sherbakov, Jessica Shieh, Sarah Shoker, Pranav Shyam, Szymon Sidor, Eric Sigler, Maddie Simens, Jordan Sitkin, Katarina Slama, Ian Sohl, Benjamin Sokolowsky, Yang Song, Natalie Staudacher, Felipe Petroski Such, Natalie Summers, Ilya Sutskever, Jie Tang, Nikolas Tezak, Madeleine B. Thompson, Phil Tillet, Amin Tootoonchian, Elizabeth Tseng, Preston Tuggle, Nick Turley, Jerry Tworek, Juan Felipe Cerón Uribe, Andrea Vallone, Arun Vijayvergiya, Chelsea Voss, Carroll Wainwright, Justin Jay Wang, Alvin Wang, Ben Wang, Jonathan Ward, Jason Wei, CJ Weinmann, Akila Welihinda, Peter Welinder, Jiayi Weng, Lilian Weng, Matt Wiethoff, Dave Willner, Clemens Winter, Samuel Wolrich, Hannah Wong, Lauren Workman, Sherwin Wu, Jeff Wu, Michael Wu, Kai Xiao, Tao Xu, Sarah Yoo, Kevin Yu, Qiming Yuan, Wojciech Zaremba, Rowan Zellers, Chong Zhang, Marvin Zhang, Shengjia Zhao, Tianhao Zheng, Juntang Zhuang, William Zhuk, and Barret Zoph. 2024. Gpt-4 technical report. *Preprint*, arXiv:2303.08774.

Jingyuan Qi, Zian Jia, Minqian Liu, Wangzhi Zhan, Junkai Zhang, Xiaofei Wen, Jingru Gan, Jianpeng Chen, Qin Liu, Mingyu Derek Ma, Bangzheng Li, Haohui Wang, Adithya Kulkarni, Muhao Chen, Dawei Zhou, Ling Li, Wei Wang, and Lifu Huang. 2024. Metascientist: A human-ai synergistic framework for automated mechanical metamaterial design. *ArXiv*, abs/2412.16270.

Marissa Radensky, Simra Shahid, Raymond Fok, Pao Siangliulue, Tom Hope, and Daniel S. Weld. 2024. Scideator: Human-llm scientific idea generation grounded in research-paper facet recombination. *ArXiv*, abs/2409.14634.

Samuel Schmidgall, Yusheng Su, Ze Wang, Ximeng Sun, Jialian Wu, Xiaodong Yu, Jiang Liu, Zicheng Liu, and Emad Barsoum. 2025. Agent laboratory: Using llm agents as research assistants. In *arXiv*, volume abs/2501.04227.

Chenglei Si, Diyi Yang, and Tatsunori Hashimoto. 2024. Can llms generate novel research ideas? a large-scale human study with 100+ nlp researchers. *arXiv preprint arXiv:2409.04109*.

Robyn Speer, Joshua Chin, and Catherine Havasi. 2017. Conceptnet 5.5: An open multilingual graph of general knowledge. In *Proceedings of the AAAI conference on artificial intelligence*, volume 31.

Jonathan M. Stokes, Kevin Yang, Kyle Swanson, Wengong Jin, Andrés Cubillos-Ruiz, Nina M. Donghia, Craig R. MacNair, Shawn French, Lindsey A. Carfrae, Zohar Bloom-Ackermann, Victoria M. Tran, Anush Chiappino-Pepe, Ahmed H. Badran, Ian W. Andrews, Emma J. Chory, George M. Church, Eric D. Brown, T. Jaakkola, Regina Barzilay, and James J. Collins. 2020. A deep learning approach to antibiotic discovery. *Cell*, 181:475–483.

Qingyun Wang, Doug Downey, Heng Ji, and Tom Hope. 2023a. Scimon: Scientific inspiration machines optimized for novelty. In *Annual Meeting of the Association for Computational Linguistics*.

Ruoyao Wang, Graham Todd, Ziang Xiao, Xingdi Yuan, Marc-Alexandre Côté, Peter Clark, and Peter Jansen. 2024. Can language models serve as text-based world simulators? In *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)*, pages 1–17, Bangkok, Thailand. Association for Computational Linguistics.

Ruoyao Wang, Graham Todd, Xingdi Yuan, Ziang Xiao, Marc-Alexandre Côté, and Peter Jansen. 2023b. ByteSized32: A corpus and challenge task for generating task-specific world models expressed as text games. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 13455–13471, Singapore. Association for Computational Linguistics.

Shunyu Yao, Jeffrey Zhao, Dian Yu, Nan Du, Izhak Shafran, Karthik R Narasimhan, and Yuan Cao. 2023. React: Synergizing reasoning and acting in language models. In *The Eleventh International Conference on Learning Representations*.

Zheng Yuan, Hongyi Yuan, Chuanqi Tan, Wei Wang, and Songfang Huang. 2023. How well do large language models perform in arithmetic tasks? *ArXiv*, abs/2304.02015.

Appendix

A Table of Contents

Below is a list of links to major sections:

B External Reviewer Rubric

External reviewers were provided with each of the papers from Table 4, and asked to categorize them on 2 categorical scales: scientific soundness and novelty. In addition, they were asked to provide short justification for their ratings, as well as an overall brief description of the contributions and claims of the paper.

Soundness refers to the rigor and reliability of a study's methods and evidence—essentially, how well the experiments and analysis support the claims made. Please provide a rating for the **soundness** of this study:

(A) Clearly Sound: The study demonstrates robust methodology; its design, implementation, and analysis fully support the claims.

(B) Likely Sound: Assuming faithful execution, the methodology appears sound, with evidence generally supporting the claims despite minor uncertainties.

(C) Minor Concerns: Identified methodological limitations may slightly affect measurements (e.g., effect sizes) but do not alter the overall conclusions.

(D): Unsound: Evident methodological or conceptual flaws undermine the credibility of the claims and contributions.

Novelty refers to how original or innovative a study's contributions are relative to existing work. It assesses whether the contributions offer incremental changes or significant departures from what has been done before. Please provide a rating for the **novelty** of this study.

(A) Highly Novel: Introduces entirely new concepts or frameworks not previously explored. Example: A modeling contribution that proposes a novel architecture that redefines established paradigms in NLP.

(B) Incrementally Novel (Significant Variation): Substantially modifies existing approaches, leading to marked advancements. Example: A modeling contribution that makes significant architectural or algorithmic changes that enhance performance.

(C) Incrementally Novel (Minor Variation): Presents modest modifications or adaptations to established work. Example: A modeling contribution that applies an existing model to a new task with only minor tweaks or parameter adjustments.

(D): Not Novel/Exists in Exact Form: Replicates existing work without introducing any modifications. Example: A modeling contribution that runs an existing model on an existing task, where the result is already known.

C Meta-Analysis Categorization Criteria

The following criteria were used to classify a suite of 5 experiments as having either **consistent**, **mixed**, or **limited** results:

1. **Consistent (C):** If at least 80% (i.e. 4 of the 5) independent experiment runs for a given

idea generated the same high-level result (i.e. supporting or rejecting the hypothesis), then it was classified as consistent.

2. **Mixed (M):** If a set of 5 experiments was neither classified as *consistent* or *limited*, then it was classified as having *mixed* results.

3. **Limited (L):** If 40% or fewer (i.e. 2 or fewer of the 5 runs) successfully completed, regardless of the outcome of those experiments (i.e. support, reject, or inconclusive towards the hypothesis), then it was classified as limited.

D Example Domain-Expert Comments

The domain expert comments are appended to each selected idea before the planning stage, typically to correct minor issues (such as selecting a better metric), or clarify portions of the idea. Pilot experiments without these comments showed that the experiments generally still work, but have less-strong conclusions (e.g. a less robust metric will be used, such a direct string matching instead of a more robust LLM-as-a-judge metric, limiting what can be claimed), or may not find results at all (e.g. the system may wish to measure an agent's success only by task completion, which is often zero for most agents and environments – whereas instead using the normalized partial progress scores offered by most environments is standard practice). To highlight the limited nature of these comment, the expert comments for each of the 19 discoveries in Table 4 are provide in Table 6

E Example Idea, Comments, and Plan

An example automatically generated idea, set of domain-expert comments, and the resulting generated plan from the combination of the two are provided in Table 7.

#	Domain Expert Comment
Candidate discoveries identified by external reviewers, and confirmed by internal review	
1	**State Prediction Confidence:** Measuring prediction accuracy could be done using LLM-as-a-judge (e.g. have the model predict the observation, then have another LLM compare this generated observation to the gold observation, counting (perhaps by sentence, or by item) the number of things that are the same, and the number that are different, arriving at a score between 0-1 for each state prediction. Similarly, do to the task well, the LLM doing the state prediction task should probably have at least the last 2-3 observations/actions in its prompt, to provide some context.
2	**Accuracy vs Representational Expressivity:** No additional comments provided.
3	**Multi-Stage Environment Generation:** Solid idea – try to build games incrementally rather than in one-shot, to see if that improves performance. Doesn't mention where the source templates come from (presumably ideated from ByteSized32, so likely from that corpus/benchmark – though it could also try to build them from scratch, or from a simple predefined template that it builds for this task, to make it easier). It's also proposing to use a regex-based checker for game mechanics rather than the ByteSized32 evaluation methods – that might work, or it might require an LLM-as-a-judge situation if the regex matching is not successful. (Could include both in the evaluation, and compare them).
4	**Combinatorial Optimization:** Could be interesting to see if an LLM can do this as well as a simple mathematical solver. Should include a notion of tolerance (not in terms of the resistor tolerance, like 1%, 5%, etc., but in how close the value the different solvers create have to be to the real value – otherwise some solutions may not be possible). Should have a check that verifies the solutions (from the LLM, and other solvers) are within (say) 1% or 5% or 10% of the expected value (or, could use all three of these, as a sort of graded accuracy metric).
5	**Action Prediction:** Kind of makes sense, and would be interesting to see. While the specification says to just provide a binary prediction (yes/no) as to whether the action will succeed (as well as the confidence score), it's not super clear what 'action will succeed' means. Does it means the action will run in the interpreter? (in which case, it's not super interesting because, as long as the action is in the valid action list, it should run). More interesting would be if it interpreted some signal that it worked (e.g. you can't cook a fridge or chop a pot, and the environment might say this, then (using a cheap LLM call), you might be able to interpret whether the observation returned after the action signified success or failure (e.g. 'you can't do that')). But, extending this, it'd be interesting if it predicted more than binary success, but also did more of a state-prediction task – e.g. predicting what the next observation will be, and then using an LLM to verify how much of it is essentially correct (perhaps proportion of sentences correct). It'd need some number of steps of past history (say the last 1, 2, or 3 steps) to have a chance at doing this well.
6	**Graph Agent for Discovery:** Might be hard to get the DiscoveryWorld knowledge score working at the start (and extracting this coherently from the agent's memory) – I'd focus on the DiscoveryWorld Task Completion and (more importantly) Task Process scores.
Results rejected by either external reviewers, internal review, or both	
7	**Social Graphs:** This could work – but depends very much on the complexity and challenges required in interacting the social relationships. It sounds like this proposes to create the benchmark rather than use an existing one – so it would need to make sure that the interactions are interesting, reasonably complex, and non-trivial to navigate. It'd also need some clear measure of evaluating an agent's performance – it's not clear what 'accuracy of relationship-based decisions' is or how it would be measured.
15	**Planning Agent:** Mostly makes sense, but one of its assumptions (focusing on get/put/cook recipes) isn't possible, it'd have to change this – there's no way of limiting what actions need to be used. Also it should use the task score, not task completion rate. Most agents do not complete any tasks, but the task score is a partial score between 0 and 1 that is often non-zero if an agent makes task progress.
9	**Spatial Agent:** Makes a lot of sense, and you'd expect this to work. Somewhat related to other agents (though I'm not sure any have tried augmenting ReAct in this way, or on this environment). Should use the partial task score instead of task completion rate as a measure of success – the tasks are hard and most agents don't complete them, but the partial task score gives a score between 0-1 that measures partial progress.
10	**Action History:** Makes sense. There have been a lot of similar ideas generated, the one that makes this one more viable is that it's not just tracking successful actions in isolation, but considering the *context* in which they occurred. Text games generally require long action sequences, where each action is taken at the appropriate time, when all the right conditions are met. Taking the context into account should help it figure out when it's appropriate to take a particular action. Progress should be measured using the Task Score (a measure of partial progress), not the Task Success Rate, since task success is rare with most agents in these environments.
11	**Template Agent:** Makes sense, but (1) should use increasing partial task score (0-1), rather than task success/completion, as a signal – since this environment is hard, and task success is rare.
12	**Graph Verification Agent:** No additional comments provided.
13	**Affordance Agent:** Makes sense – uses an LLM to predict affordances, then act based on those affordances. Perhaps could augment a ReAct agent with 3 steps (affordances, think, act) rather than just the normal 2 (think/act). Should measure performance in terms of the task score, rather than task success (since task success is rare). Could use the 'find living thing' subtask (one of the easiest ones) as an additional task to try.
14	**WordNet:** Super simple baseline agent. One might expect it to outperform a random baseline. It might even outperform a ReAct agent on this task, especially if paired with aspects that allow the agent to explore the environment. Should likely be run on very simple (e.g. 3 room maximum) environments.
15	**Metaphor Graphs:** At first glance it's hard to see how metaphors would be useful here, but the suggested operationalization (e.g. 'what functional similarities exist between X and Y in a cooking context?') might help it better organize the graph into categories of objects. The "project supervisor" ratings (i.e. manual human ratings) should likely not be included, since this requires human ratings, and interrupts the automatic flow of running the experiment.
16	**Goal Tracking:** It might work, though it doesn't mention a base agent (like ReAct) to augment with the goals. It's good that it mentions using task score (rather than task completion) as a metric, since task completion is often zero for these hard tasks, where as task score is often non-zero if the agent is making some progress.
17	**Container Agent:** It's specifically focused on building a graph of relevant container relationships, which at first seemed uninteresting, but now that I think about it, basic e.g. ReAct agents tend to struggle with finding ingredients – so having a graph of where they tend to be could help it. Same for tools it needs (e.g. cooking implements, recipe book, knife for chopping, etc.). Presumably the graph would be included in the ReAct agent prompt. The metric should not be task completion (since task success is hard and rarely non-zero on this task), but rather the task score, which provides a partial measure of task progress (with a value between zero and one).
18	**Template-based Environment Creation:** Mostly makes sense, but one of its assumptions (focusing on get/put/cook recipes) isn't possible, it'd have to change this – there's no way of limiting what actions need to be used. Also it should use the task score, not task completion rate. Most agents do not complete any tasks, but the task score is a partial score between 0 and 1 that is often non-zero if an agent makes task progress.
19	**Graph Metric:** Neat idea (and, very different than many of the others). Would benefit from using vector similarity.

Table 6: Domain expert comments provided at the ideation stage, for each of the 19 potential discoveries from Table 4.

Field	Content
Ideation	
name	simulation-confidence-analysis
long_description	Study whether LLMs can accurately assess their confidence in state predictions, and whether this confidence correlates with actual accuracy. This could enable more reliable simulation by identifying when predictions are likely to be incorrect.
short_description	Investigate LLM ability to assess confidence in state predictions and correlation with accuracy.
hypothesis	LLM confidence scores will correlate with prediction accuracy, allowing for identification of potentially incorrect predictions.
variables	Independent variables: State complexity, Game type, Property type. Dependent variables: Prediction accuracy, Confidence score. Control: Same LLM, same states, same examples.
metric	Correlation between confidence scores and accuracy. Precision/recall for identifying incorrect predictions using confidence thresholds.
pilot	Test on simple CookingWorld scenarios, focusing on boolean property predictions with confidence scores.
example_prompt	Create an experiment to analyze LLM confidence in state predictions. Use TextWorldExpress to generate 200 state transitions. For each prediction, prompt GPT-4 to provide both the predicted state and a confidence score (0–100) for each property change. Log all predictions, confidence scores, and ground truth. Calculate correlation between confidence and accuracy. Generate ROC curves for using confidence to predict correctness. Use bootstrap resampling to compute confidence intervals. Create visualizations showing the relationship between confidence and accuracy across different property types.
Human Rating	
human_rating	potentially feasible
human_notes	Measuring prediction accuracy could be done using LLM-as-a-judge (e.g. have the model predict the observation, then have another LLM compare this generated observation to the gold observation, counting (perhaps by sentence, or by item) the number of things that are the same, and the number that are different, arriving at a score between 0–1 for each state prediction). Similarly, the LLM doing the state prediction task should probably have at least the last 2–3 observations/actions in its prompt to provide context.
conditioning_text	Please use `gpt-4o-mini` for all LLM calls, because it's fast and inexpensive.
Operationalization	
operationalization	Please create an experiment to analyze LLM confidence in state predictions in TextWorldExpress, implementing the following specifications: **EXPERIMENT MODES AND SCOPE:** Implement a global variable `PILOT_MODE` that can be set to one of: 'MINI_PILOT', 'PILOT', or 'FULL_EXPERIMENT'. For each mode: `MINI_PILOT`: 3 episodes of CookingWorld, 10 steps each (training set); `PILOT`: 20 episodes of CookingWorld, 25 steps each (training set); `FULL_EXPERIMENT`: 200 episodes, 50 steps each (balanced across train/dev/test sets). **ENVIRONMENT SETUP:** Use the TextWorldExpress API to generate CookingWorld scenarios with simple environments (3 rooms, no doors, 2 ingredients, 2 distractor items). Record the current state, action, and next state at each step. **LLM CONFIGURATION:** Use `gpt-4o-mini` for all LLM calls. The state prediction prompt should include the last 2 observations, the current action, a request for the next state prediction, and a confidence score (0–100) for each property change. **DATA COLLECTION PROCEDURE:** For each step, retrieve the current state and action; prompt the LLM for a state prediction with confidence scores; retrieve the actual next state; and use an LLM-as-a-judge to score prediction accuracy (score: 0–1 per property change). **DATA ANALYSIS:** For each episode, calculate the correlation between confidence scores and accuracy, and generate scatter plots, ROC curves, and visualizations of the confidence–accuracy relationship. **LOGGING AND OUTPUT:** Log all raw data (states, predictions, confidence scores, accuracy scores), generate summary statistics, and create plots (scatter plot, ROC curves, box plots). **EXECUTION FLOW:** Run `MINI_PILOT` first; if successful, run `PILOT` and then stop. **SUCCESS CRITERIA:** Achieve clean execution in `MINI_PILOT` and find meaningful correlation patterns in `PILOT`, with statistical significance verified via bootstrap resampling.
code_examples	TextWorldExpress API Example, Non-parametric Bootstrap Resampling, Logger/Debugging, MatPlotLib Line Plot, LLM example through proxy server

Table 7: An example idea, the domain expert comments, and the generated plan (produced by using both the idea and comments).

F Prompts

Core prompts are provided below.

Ideation Prompt

```
You are ScientistGPT, the most advanced automated scientific model in the world. You can use your enormous intellect to solve any problem, and the solutions to these problems may help improve our knowledge of how the world works, which is a noble and important goal.
You are currently working on the following task: Generating new research ideas/ideas for new experiments to run.
The goal of running the experiments is to generate novel, interesting, and (ideally) high-impact scientific results.
Below is a set of scientific research papers (expressed as their Latex source).
Your task is to come up with new research ideas, and follow-on research ideas, based on the research questions, research programs, hypotheses, operationalizations of experiments, or any other information provided in these papers.
You are asked to come up with 5 ideas.
You can use content from one paper, or combine content from multiple papers to generate new ideas.
The ideas you generate can be highly novel inspired by the research in the papers below, or they can be incremental follow-on ideas based on the papers below. The most important thing is that we're doing good, reasoned, and potentially impactful/useful science.
Some recipes for ideation you might use are the following:
1. **Filling the gaps**: Identify gaps (high-level or low-level) in the research programs below, and come up with ideas to fill those gaps.
2. **Abstractive**: Abstract the research programs below to a higher level, and come up with new ideas based on those abstractions.
3. **Combining ideas**: Combine ideas from different research programs below to come up with new ideas.
4. **Extending ideas**: Extend the ideas from the research programs below to come up with new ideas.
5. **Challenging assumptions**: Challenge the assumptions made in the research programs below, and come up with new ideas based on those challenges.
6. **What happens if**: Come up with ideas that ask what happens if you change key/important parts of the research programs below.
7. etc.

After reading the research papers (and their implicit or explicit Research Programs, Hypotheses, and Operationalizations of Experiments contained within the papers) below, you will be asked to come up with a list of new research ideas (which can be highly novel or incremental follow-on ideas).
As a strategy, you can try coming up with one idea for *each* of the methods above (i.e. filling the gaps, abstractive, combining ideas, extending ideas, challenging assumptions, etc.), or subsampling this if you need to generate fewer ideas.
The response format (JSON) is below:
```json
[   # List of research ideas
    {    # Research Idea 1
         "research_idea_name": "A 2-3 word name of this research idea, hypthen-separated (e.g. my-research-idea)",
         "research_idea_long_description": "A long (e.g. ~50-100 word) description of the research idea, and what it's investigating.",
         "research_idea_short_description": "A short (e.g. ~20 word) high-level description of the research idea, and what it's investigating.",
         "research_idea_hypothesis": "What is the hypothesis of the research idea?  What are you trying to prove or disprove?",
         "research_idea_variables": "What are the main variables involved in investigating this research program?  What variables are held constant, and what variables are manipulated?",
         "research_idea_metric": "What is the main metric that will be used to evaluate the success of this research idea? How will we know if the idea works or not?  How will partial performance be measured?",
         "research_baselines": "If your system is an experimental system, what baselines will you compare against?  I.e. if you're creating a (a) new method based on an old method, or a (b) modification of an existing method, then you should probably compare to the old method or the existing method.  If you're creating a new method from scratch, then you should probably compare to a simple method that is easy to beat, or a method that is similar to yours in some way.",
         "research_idea_pilot": "What's the simplest version of this that can be tested, before running a more expensive version?  Usually this is a full (or reasonably full) method, but on a small subset of the input data.",
         "research_idea_design_prompt": "Provide a detailed design of the experiment here, with enough detail that it can be implemented by a student-level practitioner (which in actuality, is an automated experiment building system).  This is the only text they will be provided to build the experiment, so be specific.  DO NOT JUST GIVE HIGH-LEVEL DESCRIPTIONS (like 'implement the experiment') because this is useless.  This should minimally include at least: (1) *Detailed* mid-level descriptions of what is to be implemented, including any algorithms (designed at a high or low level), (2) Detailed descriptions of what data to use, in the context of a pilot experiment, (3) Detailed descriptions of what output to generate, how to save it (in the context of maximum utility for follow-on experiments), and how to evaluate and report the results."
         "research_idea_codeblocks": ["A list of existing codeblocks from the codeblock library that this idea is likely to use"],
         "research_idea_required_code_and_resources": [ # An EXHAUSTIVE list of ALL required CODE, RESOURCES, MODELS, etc. mentioned in this ENTIRE RESERACH IDEA.  CRITICALLY IMPORTANT, USED TO DETERMINE FEASIBILITY! If it's mentioned above, it ABSOLUTELY NEEDS to be here!
             {"name": "example short name", "description": "a short example description of the code or resource that is needed", "where": "one of: `existing codeblock`, `external`, or `build`", "effort": "one of: `minor`, `moderate`, or `major`"}, # `where` refers to where the code/resource/model comes from (an existing codeblock template, an external source that can be retrieved, or whether we need to build it for this work.  `effort` refers to how much effort that process will take: `minor` (e.g. small modifications/trivial code), `moderate` (a good amount of work), and `major` for large and/or high-difficulty volumes of code.
             {"name": "ReAct baseline", "description": "A ReAct baseline (targeted for use on Benchmark XYZ)", "where": "existing codeblock", "effort": "minor"}, # `existing codeblock` because there's an existing codeblock covering a ReAct baseline, and `minor` because this is just using the existing codeblock.
             {"name": "Modified ReAct baseline", "description": "The proposed modified ReAct model", "where": "existing codeblock", "effort": "moderate"}, # `existing codeblock` because there's an existing codeblock covering a ReAct baseline, and `moderate` because this is proposing non-trivial (but not huge) modifications to that agent, so it will take some work to build.
```
```

```
            {"name": "Benchmark X", "description": "The primary benchmark used to evaluate the models", "where": "existing codeblock", "effort": "minor"},  # `existing codeblock` because this benchmark has an existing codeblock covering it, that just needs to be directly applied
            {"name": "Benchmark Y", "description": "The secondary (transfer) benchmark used to evaluate the models", "where": "external", "effort": "moderate"},    # `external` because there's no existing codeblock for this specific benchmark. `moderate` because this isn't a popular benchmark available on e.g. huggingface, so it will likely take a bit of work to find/download/load it.
            {"name": "LLM interface", "description": "The interface to prompt the LLM for the agents", "where": "existing codeblock", "effort": "minor"}, # The agents need LLM calls. This is an existing codeblock with minor modifications (if any).
            {"name": "gpt-4o model", "description": "The gpt-4o model available from the OpenAI API", "where": "existing codeblock", "effort": "minor"}, # The base model to use for the agents. The LLM codeblock covers using it, so we don't need to download it, and it should be low effort.
            {"name": "Prior Agent ABC", "description": "An existing agent described in the paper, to serve as a secondary baseline", "where": "external", "effort": "major"}, # `external` because we don't have a codeblock for it and it's something someone else published on github. `moderate` since it's usually a fair amount of work getting someone else's agent working.
            {"name": "Fancy New Agent++", "description": "A new agent proposed in this work integrating ...", "where": "build", "effort": "major"}, # `build` because we have to largely build it from scratch, and `major` because it's a fairly complex new agent algorithm that is likely to take a lot of work to build.
            {"name": "Bootstrap resampling", "description": "The bootstrap resampling technique for comparing the performance of two models", "where": "existing codeblock", "effort": "minor"}, # Already compltely covered in an existing codeblock, we just have to use it
            {"name": "New dataset collection", "description": "A new dataset collection procedure for collecting data for the agents through web scraping", "where": "build", "effort": "major"}, # `build` because we have to build it from scratch, and `major` because it's a fairly complex new data collection procedure that is likely to take a lot of work to build and debug.
            # ... More code/resources/models/etc, if any
        "research_idea_external_requirements": ["An exhaustive list of libraries or packages that may be required. Format: `python/apt package name (very short description of need)`", "transformers (for XYZ)", "scikit-learn (for ABC)", ...]
    },
    ... # More research ideas
]
```

IMPORTANT NOTE: An exhaustively detailed and complete `research_idea_required_code_and_resources` is *ABSOLUTELY REQUIRED*, as this is used to prepare the experiment workspace, and determine experiment feasibility. A poorly or incorrectly documented `research_idea_required_code_and_resources` for an idea is a major failure, as it will waste a large amount of resources (time/money/etc) on ideas that may be unlikely to have the resources they need to succeed.

NOTE: Below is a simple example `research_idea_design_prompt` for a hypothetical example research idea:
```
Please create an agent that automatically builds an informative, useful knowledge graph from exploring its environment. The knowledge graph should be expressed as triples, i.e. subject-relation-object, and stored in DOT/Graphviz format. A knowledge graph should be saved at each step, so we can see how they evolve. The graphs should be converted from DOT to PDF so the user can view them, with the 'new' nodes highlighted in a different color (and these should be in the report, when you get to this stage). Please test this on CookingWorld, using the default CookingWorld environment parameters (except 3 rooms, and no doors). The base model should be `gpt-4o-mini`. The agent should spend the first 10 steps of each episode exploring, primarily to build the knowledge graph. It should then spend the remaining steps alternating between 'explore' (knowledge building) and 'exploit' (using the knowledge in the knowledge graph to perform some relevant action that makes progress towards the goal). The agent should use the first 2 parametric variations (i.e. the first three episodes, seeds 1-2) of the CookingWorld game, storing one knowledge graph per episode of the game. The maximum steps per episode should be 40. The full trajectory (i.e. observation, score, possible valid actions, chosen action at each step) should be in the log file.
```

You are asked to generate new research ideas that are *conditioned*/*related to* the kinds of codeblocks that the automated experiment builder has available in the codeblock library. Here are high-level summaries of the code templates available in the experiment builder:
```
<Insert codeblock summaries here>
```

Existing Research Papers, from which you should consider their (implicitly or explicitly stated) Research Programs, Hypotheses, and Operationalizations of Experiments:
Example Paper 1:
```
<Insert Latex of Paper 1 Here>
```

Example Paper 2:
```
<Insert Latex of Paper 2 Here>
```

Please generate a list of new research ideas (which can be highly novel or incremental follow-on ideas). The most important thing is that we're doing good, reasoned, and potentially impactful/useful science.
After reading the research papers (and their implicit or explicit Research Programs, Hypotheses, and Operationalizations of Experiments contained within the papers) below, you will be asked to come up with a list of new research ideas (which can be highly novel or incremental follow-on ideas).
You are asked to come up with 5 ideas.

```
As a strategy, you can try coming up with one idea for *each* of the methods above (i.e. filling the gaps, abstractive,
 \  combining ideas, extending ideas, challenging assumptions, etc.), or subsampling this if you need to generate fewer
 \  ideas.
The response format (JSON) is below:
```json
[   # List of research ideas
    {    # Research Idea 1
         "research_idea_name": "A 2-3 word name of this research idea, hypthen-separated (e.g. my-research-idea)",
         "research_idea_long_description": "A long (e.g. ~50-100 word) description of the research idea, and what it's
          \  investigating.",
         "research_idea_short_description": "A short (e.g. ~20 word) high-level description of the research idea, and what
          \  it's investigating.",
         "research_idea_hypothesis": "What is the hypothesis of the research idea?  What are you trying to prove or
          \  disprove?",
         "research_idea_variables": "What are the main variables involved in investigating this research program?  What
          \  variables are held constant, and what variables are manipulated?",
         "research_idea_metric": "What is the main metric that will be used to evaluate the success of this research idea?
          \  How will we know if the idea works or not?  How will partial performance be measured?",
         "research_baselines": "If your system is an experimental system, what baselines will you compare against?  I.e.
          \  if you're creating a (a) new method based on an old method, or a (b) modification of an existing method, then
          \  you should probably compare to the old method or the existing method.  If you're creating a new method from
          \  scratch, then you should probably compare to a simple method that is easy to beat, or a method that is
          \  similar to yours in some way.",
         "research_idea_pilot": "What's the simplest version of this that can be tested, before running a more expensive
          \  version?  Usually this is a full (or reasonably full) method, but on a small subset of the input data.",
         "research_idea_design_prompt": "Provide a detailed design of the experiment here, with enough detail that it can
          \  be implemented by a student-level practitioner.  This is the only text they will be provided to build the
          \  experiment, so be specific.  This should minimally include at least: (1) Detailed descriptions of what is to
          \  be implemented, including any algorithms (designed at a high or low level), (2) Detailed descriptions of what
          \  data to use, in the context of a pilot experiment, (3) Detailed descriptions of what output to generate, how
          \  to save it (in the context of maximum utility for follow-on experiments), and how to evaluate and report the
          \  results."
         "research_idea_codeblocks": ["A list of existing codeblocks from the codeblock library that this idea is likely
          \  to use"],
         "research_idea_required_code_and_resources": [ # An EXHAUSTIVE list of ALL required CODE, RESOURCES, MODELS, etc.
          \  mentioned in this ENTIRE RESERACH IDEA.  CRITICALLY IMPORTANT, USED TO DETERMINE FEASIBILITY! If it's
          \  mentioned above, it ABSOLUTELY NEEDS to be here!
              {"name": "example short name", "description": "a short example description of the code or resource that is
               \  needed", "where": "one of: `existing codeblock`, `external`, or `build`", "effort": "one of: `minor`,
               \  `moderate`, or `major`"}, # `where` refers to where the code/resource/model comes from (an existing
               \  codeblock template, an external source that can be retrieved, or whether we need to build it for this
               \  work.  `effort` refers to how much effort that process will take: `minor` (e.g. small
               \  modifications/trivial code), `moderate` (a good amount of work), and `major` for large and/or
               \  high-difficulty volumes of code.
              {"name": "ReAct baseline", "description": "A ReAct baseline (targeted for use on Benchmark XYZ)", "where":
               \  "existing codeblock", "effort": "minor"}, # `existing codeblock` because there's an existing codeblock
               \  covering a ReAct baseline, and `minor` because this is just using the existing codeblock.
              {"name": "Modified ReAct baseline", "description": "The proposed modified ReAct model", "where": "existing
               \  codeblock", "effort": "moderate"}, # `existing codeblock` because there's an existing codeblock covering
               \  a ReAct baseline, and `moderate` because this is proposing non-trivial (but not huge) modifications to
               \  that agent, so it will take some work to build.
              {"name": "Benchmark X", "description": "The primary benchmark used to evaluate the models", "where":
               \  "existing codeblock", "effort": "minor"},  # `existing codeblock` because this benchmark has an existing
               \  codeblock covering it, that just needs to be directly applied
              {"name": "Benchmark Y", "description": "The secondary (transfer) benchmark used to evaluate the models",
               \  "where": "external", "effort": "moderate"},    # `external` because there's no existing codeblock for
               \  this specific benchmark. `moderate` because this isn't a popular benchmark available on e.g. huggingface,
               \  so it will likely take a bit of work to find/download/load it.
              {"name": "LLM interface", "description": "The interface to prompt the LLM for the agents", "where": "existing
               \  codeblock", "effort": "minor"}, # The agents need LLM calls. This is an existing codeblock with minor
               \  modifications (if any).
              {"name": "gpt-4o model", "description": "The gpt-4o model available from the OpenAI API", "where": "existing
               \  codeblock", "effort": "minor"}, # The base model to use for the agents. The LLM codeblock covers using
               \  it, so we don't need to download it, and it should be low effort.
              {"name": "Prior Agent ABC", "description": "An existing agent described in the paper, to serve as a secondary
               \  baseline", "where": "external", "effort": "major"}, # `external` because we don't have a codeblock for
               \  it and it's something someone else published on github. `moderate` since it's usually a fair amount of
               \  work getting someone else's agent working.
              {"name": "Fancy New Agent++", "description": "A new agent proposed in this work integrating ...", "where":
               \  "build", "effort": "major"}, # `build` because we have to largely build it from scratch, and `major`
               \  because it's a fairly complex new agent algorithm that is likely to take a lot of work to build.
              {"name": "Bootstrap resampling", "description": "The bootstrap resampling technique for comparing the
               \  performance of two models", "where": "existing codeblock", "effort": "minor"}, # Already compltely
               \  covered in an existing codeblock, we just have to use it
              {"name": "New dataset collection", "description": "A new dataset collection procedure for collecting data for
               \  the agents through web scraping", "where": "build", "effort": "major"}, # `build` because we have to
               \  build it from scratch, and `major` because it's a fairly complex new data collection procedure that is
               \  likely to take a lot of work to build and debug.
              {"name": "Cohen's Kappa", "description": "The Kappa measure of interannotator agreement for the dataset (use
               \  the `sklearn` library implementation)", "where": "build", "effort": "minor"}, # Fairly straightforward
               \  use of this library, so it's a minor effort.
              {"name": "Rouge score", "description": "The Rouge score for evaluating the quality of the generated text (use
               \  the `rouge-score` library implementation)", "where": "existing codeblock", "effort": "minor"}, # Already
               \  covered in an existing codeblock, so it's a minor effort.
              # ... More code/resources/models/etc, if any
```
```

```
        "research_idea_external_requirements": ["An exhaustive list of libraries or packages that may be required.
         \  Format: `python/apt package name (very short description of need)`", "sklearn (for kappa)", "rouge-score
         \  (for rouge score)", ...]
     },
     ... # More research ideas
]
```

IMPORTANT NOTE: An exhaustively detailed and complete `research_idea_required_code_and_resources` is *ABSOLUTELY
 \ REQUIRED*, as this is used to prepare the experiment workspace, and determine experiment feasibility. A poorly or
 \ incorrectly documented `research_idea_required_code_and_resources` for an idea is a major failure, as it will waste a
 \ large amount of resources (time/money/etc) on ideas that may be unlikely to have the resources they need to succeed.

NOTE: Below is a simple example `research_idea_design_prompt` for a hypothetical example research idea:
```
Please create an agent that automatically builds an informative, useful knowledge graph from exploring its environment.
 \  The knowledge graph should be expressed as triples, i.e. subject-relation-object, and stored in DOT/Graphviz format. A
 \  knowledge graph should be saved at each step, so we can see how they evolve. The graphs should be converted from DOT
 \  to PDF so the user can view them, with the 'new' nodes highlighted in a different color (and these should be in the
 \  report, when you get to this stage). Please test this on CookingWorld, using the default CookingWorld environment
 \  parameters (except 3 rooms, and no doors). The base model should be `gpt-4o-mini`. The agent should spend the first 10
 \  steps of each episode exploring, primarily to build the knowledge graph. It should then spend the remaining steps
 \  alternating between 'explore' (knowledge building) and 'exploit' (using the knowledge in the knowledge graph to
 \  perform some relevant action that makes progress towards the goal). The agent should use the first 2 parametric
 \  variations (i.e. the first three episodes, seeds 1-2) of the CookingWorld game, storing one knowledge graph per
 \  episode of the game. The maximum steps per episode should be 40. The full trajectory (i.e. observation, score,
 \  possible valid actions, chosen action at each step) should be in the log file.
```
NOTE: You should try to define important terms, as the paper is likely to be unavailable to the automated experiment
 \ builder, only the information you provide will be. Similarly, acronyms can be used, but they should be defined on
 \ their first use.

Your JSON response must be between code blocks (```). You can write any other text you wish before or after (such as if
 \ you want to describe the research programs, hypotheses, and/or operationalizations of experiments in the papers), but
 \ only JSON text between a single set of codeblocks (```) will be able to be automatically extracted and used.
Remember that your research ideas should be ACTUALLY IMPLEMENTABLE by being conditioned on the kinds of codeblocks that
 \ the automated experiment builder has available. Your templates and operationalizations should particularly emphasize
 \ existing codeblocks in the experiment builder.
Similarly, while you can use external libraries or packages, you may wish to minimize the use of these, as they may not be
 \ available to the automated experiment builder, or (more likely) it may not be fluent in their use.
```

Listing 1: Ideation Prompt
```

Planning Prompt

````
You are ScientistGPT, the most advanced automated scientific model in the world. You can use your enormous intellect to
 \  solve any problem, and the solutions to these problems may help improve our knowledge of how the world works, which is
 \  a noble and important goal.
You are currently working on the following task: Converting a high-level idea for an experiment that you generated (based
 \  on reading scientific articles) into a very specific prompt to give an experiment building agent, to build and run
 \  that experiment according to your detailed specifications.
The experiment building agent is template-based -- that is to say, it (as much as possible) tries to use existing code
 \  templates (called codeblocks) to build the experiment.  This is to help reduce errors in the implementation process,
 \  as well as reduce the opportunity for scientific/resaerch methods errors.
Below is the high-level experiment idea that you generated.  Now, you must design a prompt for the experiment builder
 \  system that captures this.

Your experiment idea to convert into a prompt for the experiment builder is the following:
(Note that information provided in the idea here may not be completely accurate or usable as-is -- for example,
 \  operationalizing the idea may require more or different codeblock templates than what are mentioned in the idea below
 \  (if the idea even suggests code blocks).  Operationalizing a high-level idea often requires making changes or additions
 \  to take the idea from high-level concept to specific, implementable experiment. Please use your best judgement.)
```
<Insert research idea here>
```

In addition, you are asked to use the following SPECIAL CONDITIONING INSTRUCTIONS, that usually help re-scope the
 \  experiment to be more implementable, scope the experiment to be more towards user goals, and/or help reduce errors in
 \  the implementation process:
```
<Insert any special conditioning instructions, such as:>
Please use `gpt-4o-mini` for all LLM calls, because it's fast and inexpensive.
```

Your output must contain two keys: `prompt` and `codeblocks`.  The `prompt` key will contain the detailed prompt for the
 \  experiment builder, and the `codeblocks` key will contain a list of codeblocks that are used in the experiment.
Consequences of errors in `prompt` and `codeblocks`:
1. If the prompt is not detailed or useful, the experiment builder may not be able to build the experiment (but it will
 \  still try), and this will waste a lot of time and resources.
2. If the required codeblocks are not included, the experiment builder is highly unlikely to build the experiment
 \  successfully (but it will still try), and this will waste a lot of time and resources.

Here are two (very simple, very rough) examples of what a prompt might look like for a very simple, very hypothetical, toy
 \  experiment:
Example Prompt Generation #1:
```
{
    "prompt": "Please investigate the effect of implementing a ReAct agent with and without a small difference. In the
     \  baseline, the `think` and `act` steps of the agent should be in a single prompt (i.e. a single LLM call). In the
     \  experimental condition, the `think` and `act` steps should be in separate calls (i.e. it thinks, then it acts
     \  based on the thought). Please test this on CookingWorld, using the default CookingWorld environment parameters
     \  (except 3 rooms, and no doors). The base model should be `gpt-4o-mini`. The agent should use the first 5
     \  parametric variations (i.e. the first five episodes, seeds 1-5) of the CookingWorld game, and end after this,
     \  report the score/success of each episode, and final average score. The maximum steps per episode should be 25. The
     \  full trajectory (i.e. observation, score, possible valid actions, chosen action at each step) should be in the log
     \  file. The results file should include number of steps per episode, as well as an average of this. Report whether
     \  the baseline and experimental condition are significantly different using bootstrap resampling.",
    "codeblocks": ["Logger/Debugging", "LLM example through proxy server", "ReAct Agent Example", "TextWorldExpress API
     \  Example", "Non-parametric Bootstrap Resampling"]
}
```

Example Prompt Generation #2:
```
{
    "prompt": "Please create an agent that automatically builds an informative, useful knowledge graph from exploring its
     \  environment. The knowledge graph should be expressed as triples, i.e. subject-relation-object, and stored in
     \  DOT/Graphviz format. A knowledge graph should be saved at each step, so we can see how they evolve. The graphs
     \  should be converted from DOT to PDF so the user can view them, with the 'new' nodes highlighted in a different
     \  color (and these should be in the report, when you get to this stage). Please test this on CookingWorld, using the
     \  default CookingWorld environment parameters (except 3 rooms, and no doors). The base model should be
     \  `gpt-4o-mini`.  The agent should spend the first 10 steps of each episode exploring, primarily to build the
     \  knowledge graph.  It should then spend the remaining steps alternating between 'explore' (knowledge building) and
     \  'exploit' (using the knowledge in the knowledge graph to perform some relevant action that makes progress towards
     \  the goal). The agent should use the first 2 parametric variations (i.e. the first three episodes, seeds 1-2) of
     \  the CookingWorld game, storing one knowledge graph per episode of the game. The maximum steps per episode should
     \  be 40. The full trajectory (i.e. observation, score, possible valid actions, chosen action at each step) should be
     \  in the log file.
    "codeblocks": ["Logger/Debugging", "DOT Graphviz Graph", "LLM example through proxy server", "ReAct Agent Example",
     \  "TextWorldExpress API Example"]
}
```

*Baselines*:
If your system is an experimental system, then it's standard procedure to compare against baselines. Baselines are usually
 \  one of the following:
1. If you're creating a new method based on an old method, or a modification of an existing method, then you should
 \  probably compare to the old method or the existing method.
````

````
2. If you're creating a new method from scratch, then you should probably compare to a simple method that is easy to beat, or a method that is similar to yours in some way.
3. Sometimes, you might compare to both of the above. For example, you might have a new method (the experimental) that's a modification of an existing method (the baseline), and you might also compare to a simple method (like a random baseline) that is easy to beat.
If appropriate, please detail exactly what the baseline and experimental systems are in your prompt, how they differ, and how their performance will be meaningfully compared.

You are asked to generate an experiment prompt that is conditioned on the actual code templates available in the system, as much as possible, to help reduce the errors.  Here is a high-level summary of the codeblocks:
```
<Insert high-level summaries of codeblocks in codeblock library>
```

The following codeblocks are mentioned in the research idea, that may help you generate your experiment design prompt:

<Insert listings of codeblocks mentioned in the idea>

```



Please generate a detailed prompt for the experiment builder to construct the experiment for the idea.  That idea again is:
```
<Insert research idea here>
```

Your output must be a JSON dictionary containing two keys: `prompt` and `codeblocks`.  The `prompt` key will contain the detailed prompt for the experiment builder, and the `codeblocks` key will contain a list of codeblocks that are used in the experiment.
Your output must be a JSON dictionary between code blocks (```).  You can write any other text you wish before or after (such as if you want to describe any step-by-step thoughts you have in converting the idea into a prompt for the experiment builder), but only JSON text between a single set of codeblocks (```) will be able to be automatically extracted and used.
For example:
```
{
     "prompt": "The detailed prompt for the experiment builder goes here.",
     "codeblocks": ["List of codeblocks used in the experiment.  They must exactly match the codeblock names. If zero codeblocks are required, you must output a blank list here."]
}
```

NOTE: The codeblock names must match EXACTLY to the provided names, including capitalization, spacing, spelling, punctuation, parantheses, etc.  If they do not match exactly, the experiment builder will not be able to find the codeblocks, and the experiment will fail (at great cost).

NOTE: Please frame this as a series of pilot experiments -- so vastly reduce the amount of data/steps/etc. that are processed to just a few instances, so the experiment can be run, debugged, and verified as quickly as possible.  More details on the pilot experiment setting:
 - There should be a global variable in your code (PILOT_MODE:str) with three possible settings: `MINI_PILOT`, `PILOT`, or `FULL EXPERIMENT`.
 - The `MINI_PILOT` setting should be a very small subset of the data, and should be able to run in a few minutes.  The purpose is for fast debugging and verification of the code. For example, for question answering tasks, this might be 10 questions.  For agent tasks, this might be 2-3 episodes at 10-20 steps each.  The questions/episodes should come from the training set.
 - The `PILOT` setting should be a moderate subset of the data, ideally running in less than 1-2 hours. The purpose is to see if the results are promising, and if (for example) baseline vs experimental groups are likely to show differences.  For example, for a question answering task, this might be a few hundred questions.  For agent tasks, this might be 25-50 episodes up to 50 steps each (but this depends greatly on the task and time it takes). The questions/episodes should come from the training set for training, and the dev/validation set for evaluation, but not the unseen test set, to prevent overfitting.
 - The `FULL EXPERIMENT` setting should be the full experiment, with all data, all steps, etc.  This is the final experiment that will be run, and should be the most detailed and complete.  Training data should come from the training set.  Any hyperparamaters that need tuning should be tuned on the development set.  The experiment should be evaluated on the test set.
 - In all cases, appropriate inferrential and summary statistics should be reported, as well as any follow-on analyses. The difference between pilot levels is simply of scale, not of quality.
 - Describe the above in your experiment building prompt, so it's clear what each version should look like. - In the experiment prompt, say that it should run the MINI_PILOT first, then if everything looks good, the PILOT.  After the pilot, it should stop, and not run the FULL EXPERIMENT (a human will manually verify the results, and make the change to FULL EXPERIMENT).Please generate your JSON output now. NOTE: If you're generating newlines in your JSON strings, you must escape them properly, or they will not be parsed correctly, and the automatic extraction of your output will fail.
````

Listing 2: Planning Prompt
````

Experiment Debugging Prompt

````
You are ScientistGPT, the most advanced AI scientist and coder in the world.  You can perform any coding task, and use
 \  your enormous intellect to solve any problem correctly, systematically, and scientificially, with integrity.
Your task is to produce code that performs a specific task for a scientific experiment.  This is a reflection step -- you
 \  were previously given a task and generated code for it, which was run.  You will be shown the results, and asked to
 \  fix any errors. If everything looks good -- i.e. if the code and output meet the instruction specifications -- you'll
 \  be asked to decide that the code and execution was OK.
To support this task, you will be provided (below):
1. The instruction string from the previous task
2. Example code you were provided to generate the code
3. The code (and requirements.txt) you generated
4. The results of running the code, including any logs

Your task description for the code was the the following:
```
<Insert plan here>
```

*Change Log*
Below is the automatically generated change log, to help you know the changes that have been made along the way.  The last
 \  element is the set of most recent changes/issues.
```
<Insert Change Log here>
```


*PILOT MODE*: The requested pilot mode is: `PILOT`.  While the code should support all 3 pilot modes through a global
 \  variable (`MINI_PILOT`, `PILOT`, and `FULL_EXPERIMENT`), the pilot mode (`PILOT`) should be the one that is enabled.
 \  If it is not currently enabled in the code, please enable it. (NOTE: If there are large errors to fix in the code, you
 \  may wish to STAY AT or REVERT BACK TO `MINI_PILOT`, regardless of what the requested mode is, to make the debugging
 \  fast/inexpensive.)


*SECTION: Common code library, and code examples (called `codeblocks`)*
- You have access to a common library (`experiment_common_library`) that contains useful functions, and you can directly
 \  import them into your code.  The common library is provided below.
- In addition, you have a number of `codeblock tempates`, which are vetted code examples that (often) reference the common
 \  library.  These are not importable -- you'll need to copy or modify their code to use in your code.

*SUBSECTION: Common code library
The common library is provided below.  You can directly import these functions into your code.
```
<Insert common library here>
```

*SUBSECTION: Codeblocks (vetted code templates you should base your code on, if possible)
You have been provided with 6 template codeblocks to assist you.  They are on the following topics, with the actual
 \  codeblocks below:
<Insert list of codeblocks identified as useful for this experiment>

You should base your code AS MUCH AS POSSIBLE on these codeblocks, as (though they may look a little different than
 \  examples on the internet), they are VETTED, KNOWN-GOOD examples that you should DIRECTLY COPY as much as possible.
 \  Making errors in this environment is expensive, and using known-good code helps speed development and minimize errors.
 \  If you have to modify these codeblocks, do not hallucinate incorrect information.
The code in the codeblocks is NOT IMPORTABLE -- it is meant to be COPY AND PASTED (with whatever modifications are
 \  required) into your code.

<Insert the code listings of the codeblocks that were identified as helpful for this experiment>

*SUBSECTION: Codeblock summaries for codeblocks that were NOT picked*
Below are summaries of template codeblocks that are in the library but were NOT listed to be included in the full listings
 \  above.  If you find you need them, you can request they be included (using the `additional_codeblocks` key described
 \  below).
```
<Insert summaries of codeblocks not listed above>
```

*SECTION: Your current code and requirements*

The requirements.txt file you generated is below:
```
<Insert requirements.txt file here>
```

The code you generated is below:
```
<Insert code here>
```

*SECTION: Results of running the code*
The results of running the code are below.
*SUBSECTION: stdout, stderr, container, llm usage*
The pip stderr output is below:
```
````
````

````
<Insert pip stderr here>
```

The python stdout output is below:
```
<Insert python stdout here>
```

The python stderr output is below:
```
<Insert python stderr here>
```

The Docker errors are below:
```
<Insert container errors here>
```

Any large language model (LLM) usage by the code is below:
```
<Insert LLM usage logs (as called by the experiment code) here>
```

*SUBSECTION: Results file, and log file*
RESULTS: You should save any results (both final results, and intermediate results for further processing) in a file
 \  called `results.json`.The results file (results.json) is below:
```
<Insert results file here>
```

The log file (log.json) is below:
```
<Insert log file here>
```

*SECTION: Your current task: Reflection / Code Iteration*
You should now reflect on the code you generated, the results of running the code, and the logs.  If there are any errors,
 \  you should fix them.  If everything looks good, you should decide that the code and execution was OK.
Please provide your code below.  The output format is as follows: Your output should be between three (and exactly three)
 \  codeblocks (```).  The first codeblock will be JSON, containing a dictionary of metadata (`current_pilot_mode`,
 \  `is_ok`, `next_pilot_mode`, `issues`, `summary_of_changes`). The second codeblock will be the contents of the
 \  `requirements.txt` file, whose text will be directly copied to build a pip environment file (requirements.txt).  The
 \  third codeblock will be the contents of the `main.py` file, which is the code that will be run.  All metadata,
 \  requirements, and code must be correct and ready to run, as these will be automatically run, and not examined by
 \  humans or other processes before being automatically run.

*SUBSECTION: Writing debuggable/testable experiment code*
Here are some additional considerations when writing your experiment code:
(a) The code you're writing is scientific code to perform an experiment to test a specific hypothesis.  It should be
 \  written in a scientific, systematic, and rigorous manner, with integrity.
(b) It's very easy to make mistakes, and very hard to find them.  Your code should include *THOROUGH* checks for errors,
 \  or assumptions that may not be true.
(c) During the MINI_PILOT phase of the experiment, you should be very verbosely outputting information about the internal
 \  workings of your code to the log file, and do so in a way that is easy to understand, interpret, and spot errors in
 \  logic or assumptions. You need to make sure the code is doing what you think it's doing, and that low/high performance
 \  isn't due to a hard-to-find bug.
(d) Testing Example 1: You're sending a prompt to an LLM and expecting a response back in a certain format that's easy to
 \  parse.  But LLMs are notoriously bad at following some instructions: you need to verify (in the log file) that the
 \  response is in the correct format, and that your parser is parsing it correctly.  It's important to include checks in
 \  code that throw easily detected errors -- because seemingly minor cases (like changing the LLM prompt or base model),
 \  or running your code for longer, might expose edge cases that you didn't see earlier.
(e) Testing Example 2: You've made an agent that switches back-and-forth between different modes of operation based on
 \  some trigger.  What if that trigger never happens?  Or what if it gets stuck in one mode under certain conditions?  Or
 \  always repeats the same action?  Write your code to test for cases like these, but also output relevant information in
 \  the logs (and examine it during reflection steps) so you can notice and correct issues during debugging.

*SUBSECTION: Writing pilot experiments*
 - There should be a global variable in your code (PILOT_MODE:str) with three possible settings: `MINI_PILOT`, `PILOT`,
  \  or `FULL EXPERIMENT`.
 - The current setting of the PILOT_MODE should be whatever setting is requested by the experiment.  If no setting was
  \  explicitly requested, default to `MINI_PILOT`.
 - The `MINI_PILOT` setting should run on a very small subset of the data, and should be able to run in a few minutes.
  \  The purpose is for fast debugging and verification of the code. For example, for question answering tasks, this might
  \  be 10 questions.  For agent tasks, this might be 2-3 episodes at 10-20 steps each.  The questions/episodes should
  \  come from the training set.
 - The `PILOT` setting should be a moderate subset of the data, ideally running in less than 1-2 hours. The purpose is to
  \  see if the results are promising, and if (for example) baseline vs experimental groups are likely to show
  \  differences.  For example, for a question answering task, this might be a few hundred questions.  For agent tasks,
  \  this might be 25-50 episodes up to 50 steps each (but this depends greatly on the task and time it takes). The
  \  questions/episodes should come from the training set for training, and the dev/validation set for evaluation, but not
  \  the unseen test set, to prevent overfitting.
 - The `FULL EXPERIMENT` setting should be the full experiment, with all data, all steps, etc.  This is the final
  \  experiment that will be run, and should be the most detailed and complete.  Training data should come from the
  \  training set.  Any hyperparamaters that need tuning should be tuned on the development set.  The experiment should be
  \  evaluated on the test set.
````
````

```
 - In all cases, appropriate inferrential and summary statistics should be reported, as well as any follow-on analyses.
  \  The difference between pilot levels is simply of scale, not of quality.

*SUBSECTION: Maximum experiment runtime*
 - This experiment is run in a container.
 - The container has a user-defined maximum runtime of 7200 seconds per debug iteration.  If the experiment exceeds this
  \  runtime, it will be terminated.  Whatever files exist (e.g. logs, results, etc.) will still be reported, as of their
  \  last save.
 - If you're creating experments that are hitting the runtime limit, please consider reducing the size of the experiment.
  \  It is VERY IMPORTANT that the experiment be faithfully run -- you should favor changes that reduce the volume of data
  \  run (e.g. running on only half the training or evaluation data, and noting this) rather than modifications that
  \  change the nature, function, process, or algorithms of the experiment.

*SUBSECTION: Codeblock reminders*
 - VERY IMPORTANT: A common kind of error you make is, if you're not confident in how to implement something, you often
  \  just 'simulate' it by making code that fakes the procedure (like a fake benchmark, LLM call, algorithm, etc.). THIS
  \  IS NOT GOOD.
 - Whenever possible, the codeblock templates should be used to implement the procedures they describe.  For example, the
  \  LLM API codeblock should *always* be used to call external LLMs.  You are operating in a container, and the
  \  codeblocks in the codeblock library are VETTED to work properly for this environment.
 - Similarly, this is a scientific experiment.  Sometimes you make errors in common scientific tasks (like statistical
  \  comparisons). The codeblocks may contain VETTED, KNOWN GOOD examples of statistical comparisons.  You should ALWAYS
  \  prefer using the codeblock version of something, unless there is a strong reason otherwise.
 - If, for whatever reason, the codeblock isn't included in your list of full examples, BUT it is included in the library,
  \  you can request it be included in the full list by adding it to the `additional_codeblocks` list in the metadata.
 - The major source of error in the experiment building and debugging process is failure to find and use the codeblocks
  \  properly.  Adhering to this procedure will vastly increase the speed and accuracy of your experiment building, saving
  \  time, money, and reducing false positives/false negatives.
 - The codeblocks often use the common library (`experiment_common_library`).  Don't forget to import it, it is provided
  \  automatically in the container this code will be run in.

*SUBSECTION: Specific reminders for this task*
Remember, your code must be:
The metadata JSON dictionary should have the following format:
- `current_pilot_mode`: string.  One of `MINI_PILOT`, `PILOT`, or `FULL EXPERIMENT`.
- `is_ok_stage`: boolean.  value of `true` if you are confident the code is doing what it's supposed to do (as per the
 \  experiment instructions), and the execution is OK.  Note that the instructions might ask to implement a specific model
 \  on a specific dataset, and that model may not perform well on the dataset -- that's OK, as long as the experiment was
 \  implemented correctly and faithfully to the instructions.  The `is_ok` parameter is a check for whether the
 \  experiment was implemented correctly, not whether it performs well, or achieves interesting results.  This flag is
 \  used to signify the completion of a given experimental stage (MINI_PILOT, PILOT, FULL_EXPERIMENT).
- `is_ok`: boolean.  As above, but used to signify that the experiment is fully completed and should stop.  This should
 \  only occur when the final experiment stage has run through to completion (e.g. if the task description asks for the
 \  experiment to be run through the `PILOT` stage and stop then, this flag should be set to `true` when the PILOT stage
 \  has completed and you are confident in the results -- i.e. not if only the MINI_PILOT has run successfully.
- `next_pilot_mode`: string. One of `MINI_PILOT`, `PILOT`, or `FULL EXPERIMENT`.  What pilot mode SHOULD the experiment
 \  be running in next time?  If it's finished the current mode, this should be the next mode.  If it's not finished in
 \  the current mode, this is likely the same mode.  If there's a mode error (i.e. it should be mode X, but is actually
 \  mode Y), this should be whatever mode it *should* be in. If there are big errors to fix, you may want to revert back
 \  to MINI_PILOT to be inexpensive/fast.
- `issues`: list of strings.  Briefly describe any issues that were identified, and what their fixes are.
- `summary_of_changes`: list of strings.  Briefly describe how the code was changed to address any issues.
- `additional_codeblocks`: list of strings.  Normally an empty list.  If you need codeblocks from the codeblock library to
 \  assist in your experiment design/debugging that (for whatever reason) were not included in the initial prompt, list
 \  their names (exactly) here, and they will be included in the next debug iteration.

Remember, your code must be:
1. Correct and accurate, or it will produce wrong answers.
2. Adhere to the correct API usage, as provided in the examples, and not hallucinate or otherwise extrapolate/guess
 \  function names, or it is unlikely to work
3. Run perfectly, without error, the first time, or it will be considered a failure.
4. Run correctly without human intervention, as it will be run automatically immediately after it is generated without
 \  human review or modification.
5. Within your Python code, you should never start a line with ```, or it will mess up the automatic code extraction.
6. Never use triple-quoted strings (e.g. """) in your Python code -- they will mess up the automatic code extraction.
7. The code will be run in a container. Aside from the log files (log.json, results.json), no other files will be saved.
 \  Any files that are NOT log.json or results.json (e.g. images, figures, analyses, additional results, anything else)
 \  that the user may want MUST be saved in the `to_save/` subdirectory.  Any files in `to_save` will automatically be
 \  downloaded.  This should ideally not include large files.
8. You MUST always include exactly three (```) blocks.  The first MUST be the metadata. The second MUST be the
 \  requirements, even if it's empty.  The third MUST be the Python code.  If this isn't the case, the automatic parser
 \  will break. The codeblocks markers (```) MUST be on a newline, alone.

Remember, any errors you identify must be:
- Not hallucinated. Do not hallucinate errors that do not exist.
- Actual errors that affect the correctness of the code, data, or experiment. Do not report `errors` that are not errors,
 \  e.g., trying to make a loop more efficient, when this is not an actual error. We do not have infinite time or budget
 \  to make the code beautiful.
- Perceived Code/API errors that are not actually generating errorful behavior -- especially if these are adhering to the
 \  examples.  The example codeblocks provide known-good human-vetted implementations, and should be preferentially used
 \  in all cases except for when they are producing errors.
Example output:
The first codeblock is always metadata with the following keys: `current_pilot_mode`, `is_ok`, `is_ok_stage`,
 \  `next_pilot_mode`, `issues`, `summary_of_changes`, `additional_codeblocks`.
```
````
````

````
{
  "current_pilot_mode": "MINI_PILOT",   # Always a string
  "is_ok_stage": false   # Always a boolean
  "is_ok": false   # Always a boolean
  "next_pilot_mode": "MINI_PILOT",   # Always a string
  "issues": ["ERROR: 'numpy' is not in the requirements file"],  # Always a list of strings
  "summary_of_changes": ["Added 'numpy' to the requirements file"]   # Always a list of strings
  "additional_codeblocks": ["codeblock1name", "codeblock2name"]   # Always a list of strings
}
```
The second codeblock is always `requirements.txt`, even if empty.
```
numpy==1.21.2
```
The third codeblock is always `main.py`.
```
import numpy as np
print(np.random.rand(5))
```
````

Listing 3: Experiment Debugging Prompt
````

Reporting Prompt

````
You are ScientistGPT, the most advanced AI scientist and coder in the world.  You can perform any coding task, and use
 \  your enormous intellect to solve any problem correctly, systematically, and scientificially, with integrity.
Previously, your task was to produce code that performs a specific scientific experiment.  You wrote that code, ran it,
 \  produced a results file, and decided that the code and execution were likely OK.
Now, your task is to reflect on the goal of the experiment and results of the experiment, and write a short description of
 \  the findings in the form of a SHORT SCIENTIFIC PAPER IN LATEX.  What was the hypothesis (implicit or explicit?).  What
 \  did the results show?  Do they support or reject the hypothesis?  What are the limitations of this result?  How
 \  faithfully was the experiment that was asked for designed and tested?
You should generate tables, figures, and other scientific content as needed to support your findings.
To support this task, you will be provided (below):
1. The instruction string describing what the experiment should be testing.
2. The code (and requirements.txt) you generated
3. The results file your code generated
4. A list of any files in the `to_save/` directory, that you might want to include in your report.
5. Optionally, part (or all) of a log file that may have been generated when the experiment ran.

Your task description for the code was the the following:
```
<Insert plan here>
```

The code you generated is below:
```
<Insert code here>
```

The results file (results.json) is below:
```
<Insert results file here>
```

The files in the `to_save/` directory, and their sizes, are shown below (note, you will need to reference the code to
 \  understand what each one represents).  To use one, reference it using the filename shown below (including the relative
 \  path, e.g. `to_save/my_figure.png`):
```
<Insert list of files here>
```

The log file (log.json) is below:
```
<Insert log file here>
```

You should now reflect on the requested experiment/task, the code, the results, and the log file, and write a clear,
 \  informative, faithful, scientific, and accurate summary of the results/findings.
What was the hypothesis (implicit or explicit?).  How was it tested? What did the results show?  Do they support or reject
 \  the hypothesis?  What are the limitations of this result?  How faithfully was the experiment that was asked for
 \  designed and tested?
The report format is the complete LATEX code, that will be directly (and automatically) copy/pasted into a Latex compiler
 \  to produce the PDF, so it must be perfect the first time.
Don't forget to include tables, figures, and other scientific content as needed to support your findings.  As a general
 \  rule, if you generated the table/figure/analysis in an external file, it should probably be included in the report.
Your LATEX must be between a single set of codeblocks (```).  For example:
```
Place your complete latex code for a scientific report (similar in content to Association of Computational Linguistics
 \  (ACL) papers) here.
```
````

Listing 4: Reporting Prompt

Experiment Summary Prompt (per experiment)

````
You are ScientistGPT, the most advanced AI scientist and coder in the world.  You can perform any coding task, and use your enormous intellect to solve any problem correctly, systematically, and scientificially, with integrity.
Previously, your task was to produce code that performs a specific scientific experiment.  You wrote that code, ran it, produced a results file, and decided that the code and execution were likely OK.
Now, your task is to reflect on the goal of the experiment and results of the experiment, and write a short summary of the findings.  What was the hypothesis (implicit or explicit?).  What did the results show?  Do they support or reject the hypothesis?  What are the limitations of this result?  How faithfully was the experiment that was asked for designed and tested?
To support this task, you will be provided (below):
1. The instruction string describing what the experiment should be testing.
2. The code (and requirements.txt) you generated
3. The results file your code generated
4. Optionally, part (or all) of a log file that may have been generated when the experiment ran.

The information that you'll be asked to provide in your summary report is below:
```
- `summary`: (str) A detailed summary
- `summary_medium_detail`: (str) A medium-length summary, that is 2-3 sentences, and includes specific results (e.g. specific performance values, specific results of any statistical analyses), and a clear conclusion.
- `summary_very_short`: (str) a very short summary (maximum of 20 words)
- `hypothesis`: (str) What was the hypothesis (implicit or explicit) of the experiment?
- `hypothesis_operationalized`: (str) What was the version of the hypothesis (likely a scoped down version) that was tested through this operationalization/experiment?
- `hypothesis_inference`: (str) A clear explanation of whether the experimental results support, reject, or are inconclusive with respect to the hypothesis.
- `hypothesis_category`: (str) A string, one of `support`, `reject`, or `inconclusive`.
- `faithfullness_details`: (str) Was the experiment that was conducted a faithful representation of the experiment that was asked for?  Were there any deviations, or significant problems/errors in the implementation? and if so, what were they?
- `faithfullness_category`: (str) A string, one of `faithful`, `deviations`, or `errors`.
- `interesting_results`: (bool) Did the experiment work? And/or, were the results interesting or unexpected?  Was an experimental model significantly different than a baseline model (or, trending towards significance, in an experiment with a low number of samples)?  Set this to `true` to attract the attention of a human researcher to the results that a practitioner in the field would find interesting, and otherwise `false`.
```

Your task description for the code was the the following:
```
<Insert plan here>
```

The code you generated is below:
```
<Insert code here>
```

The results file (results.json) is below:
```
<Insert results file here>
```

The log file (log.json) is below:
```
<Insert log file here>
```

You should now reflect on the requested experiment/task, the code, the results, and the log file, and write a clear, informative, faithful, scientific, and accurate summary of the results/findings.
What was the hypothesis (implicit or explicit?).  How was it tested? What did the results show?  Do they support or reject the hypothesis?  What are the limitations of this result?  How faithfully was the experiment that was asked for designed and tested?
The summary format is a JSON dictionary with (minimally) the following keys:
- `summary`: (str) A detailed summary
- `summary_medium_detail`: (str) A medium-length summary, that is 2-3 sentences, and includes specific results (e.g. specific performance values, specific results of any statistical analyses), and a clear conclusion.
- `summary_very_short`: (str) a very short summary (maximum of 20 words)
- `hypothesis`: (str) What was the hypothesis (implicit or explicit) of the experiment?
- `hypothesis_operationalized`: (str) What was the version of the hypothesis (likely a scoped down version) that was tested through this operationalization/experiment?
- `hypothesis_inference`: (str) A clear explanation of whether the experimental results support, reject, or are inconclusive with respect to the hypothesis.
- `hypothesis_category`: (str) A string, one of `support`, `reject`, or `inconclusive`.
- `faithfullness_details`: (str) Was the experiment that was conducted a faithful representation of the experiment that was asked for?  Were there any deviations, or significant problems/errors in the implementation? and if so, what were they?
- `faithfullness_category`: (str) A string, one of `faithful`, `deviations`, or `errors`.
- `interesting_results`: (bool) Did the experiment work? And/or, were the results interesting or unexpected?  Was an experimental model significantly different than a baseline model (or, trending towards significance, in an experiment with a low number of samples)?  Set this to `true` to attract the attention of a human researcher to the results that a practitioner in the field would find interesting, and otherwise `false`.

Your output should be between codeblocks (```), and contain a single dictionary that must have the following keys.  An example of the format is below:
````

````
```
{
  "summary": "Your detailed summary here...",
  "summary_medium_detail": "Your medium-length summary here...",
  "summary_very_short": "Your very short summary here..."
  "hypothesis": "Your hypothesis here...",
  "hypothesis_operationalized": "Your operationalized hypothesis here...",
  "hypothesis_inference": "Your inference here...",
  "hypothesis_category": "Your hypothesis category here...",
  "faithfullness_details": "Your faithfullness details here...",
  "faithfullness_category": "Your faithfullness category here...",
  "interesting_results": false # true or false
}
```
````

Listing 5: Experiment Summary Prompt

Meta-Analysis Prompt

````
You are ScientistGPT, the most capable automated scientific reasoning system ever created. You can use your enormous
 \  intellect to solve any problem, and always do so in a detailed, correct, faithful way, with integrity.
Previously, you designed and ran a series of experiments centered around a particular idea/topic, though (in an effort to
 \  increase success), each implementation of that experiment was slightly different.
This is a meta-analysis step: I'll show you the results of the experiments that were run, and your job is to analyze them
 \  and draw larger-scale conclusions.

# Output
You will be asked to analyze all the experiments below, which were run based on the same original idea/topic, and provide
 \  a meta-analysis. The components of the meta-analysis are:
1. hypothesis (str): What hypothesis (either implicit or explicit) was tested by these experiments?
2. support_hypothesis_count (int): How many of the experiment runs support this hypothesis?
3. refute_hypothesis_count (int): How many of the experiment runs refute this hypothesis?
4. inconclusive_hypothesis_count (int): How many of the experiment runs are inconclusive with respect to this hypothesis?
5. detailed_summary (str): Provide a detailed natural language summary/meta-analysis of the overall results and
 \  conclusions that can be drawn from this suite of experiments.

# Idea and Operationalization/Plan
For reference, the original idea and operationalization/plan are below:
Idea:
```
<INSERT idea here>
```
Operationalization/Plan:
```
<INSERT operationalization here>
```

# Experiments
Here are the experiments that were run:
```
<INSERT experiment_results here>
```

# Output format
Please provide the following information in JSON format:
```
{
    "experiment_name": "...",
    "hypothesis": "...",
    "support_refute_inconclusive_judgements": [
        {
            "specific_experiment_name": "...",
            "brief_reasoning_for_judgement": "...",
            "judgement": "support"  # or "refute" or "inconclusive"
        },
        {
            "specific_experiment_name": "...",
            "brief_reasoning_for_judgement": "...",
            "judgement": "support"  # or "refute" or "inconclusive"
        },
        ...
    ],
    "support_hypothesis_count": 0,
    "refute_hypothesis_count": 0,
    "inconclusive_hypothesis_count": 0,
    "detailed_summary": "..."
}
```

NOTE: `experiment_name` should be the base name of the experiments. For example, if the experiment names were
 \  "my-experiment-copy1", "my-experiment-copy2", "my-experiment-copy3", etc., the base name should be "my-experiment".
SPECIAL NOTE: The "support_refute_inconclusive_judgements" field is a list of dictionaries, intended for you to make
 \  accurate, reasoned judgements about each experiment in relation to the hypothesis. It is critical that you base your
 \  judgements on an accurate, faithful interpretation of the results of each experiment relative to the stated
 \  hypothesis. Errors are very consequential.
Do not hallucinate.
Your JSON must be between code blocks (escaped as above), and must be correct, as it will be automatically extracted
 \  without human intervention. You can write any text before or after the codeblock to help you think, but the content of
 \  the codeblock must be valid JSON in the format specified above.
````

Listing 6: Meta-Analysis Prompt

G Explanations of Incremental Novelty Claims

We provide assessments of incremental novelty claims for the 6 candidate discoveries in Table 8

#	Description of Discovery and Novelty Assessment
Candidate discoveries that appear supported upon human inspection	
1	**State Prediction Confidence:** In a state prediction task, an LLM's self-assessed confidence in its predictions have a low corelation with the accuracy of those predictions. (The state prediction data was automatically crawled from one of the benchmarks) **Human Eval:** Consistent result, and to the best of our knowledge, not shown on this task. Though the correlation varies across experiments, the value consistently appears low. **Novelty Assessment:** While it is known that both (a) language models have difficulty making accurate confidence assessments in general (Geng et al., 2024), and that (b) language models have difficulty accurately performing world modeling of environments framed as state prediction (Wang et al., 2024), the combination of the two (i.e. demonstrating poor correlation between self-assessed confidence and accuracy on a world modeling tasks) in general, and the demonstration on the (self-crawled) benchmark in particular, appear incrementally novel.
2	**Accuracy vs Representational Expressivity:** In a state prediction task, an LLM performs better at predicting simpler representations (e.g. boolean values) versus states including text. (The state prediction data was automatically crawled from one of the benchmarks) **Human Eval:** Significant implementation and evaluation differences across experiments, but generally appear to support the idea that predicting simpler representations is easier. **Novelty Assessment:** While this result is intuitive (i.e. that an LLM would perform better at predicting simpler representations), this does not appear to have been demonstrated on a world modeling task framed as state prediction, and appears incrementally novel.
3	**Multi-Stage Environment Generation:** When creating novel benchmark environments using code-generation, generating the environments in multiple stages increases environment fidelity. **Human Eval:** A small change on LLM-for-environment-generation tasks, implementing specific aspects in each step, rather than generating as a whole and reflecting. For evaluation, creates a simple proxy metric that seems well-motivated as this type of evaluation is an open problem in the literature, and even llm-as-a-judge paradigms have issues with this task, while being vastly more expensive. **Novelty Assessment:** While reflection for code generation tasks is well-known, both in general (Madaan et al., 2023), and in the context of building virtual environments from templates (Wang et al., 2023b), the mechanism of explicitly and incrementally building separate categories of components in this text-game-as-code-generation task appears incrementally novel.
4	**Combinatorial Optimization:** A language model performs poorly at a combinatorial optimization problem (selecting values from a set that are closest to adding to a specified value X), grounded in substituting resistor values in electronics. **Human Eval:** Consistent result, and tested to within different tolerances, e.g. 1%, 5% **Novelty Assessment:** While the performance of language models on arithmetic problems is well studied (e.g. Yuan et al., 2023), language models are typically used to solve constraint satisfaction problems by setting up the problem, then calling an external symbolic solver (e.g. Gu et al., 2023). The particular task designed here – using a language model to substitute one resistor value (in electronics) with two or three other standard resistor values, while measuring the tolerance of that substitution – appears to be a novel task for evaluating a language model.
5	**Action Prediction:** An LLM's ability to predict whether actions will be successful in a virtual environment is generally low, marginally above a random baseline. **Human Eval:** Appears true, with the following qualifications: (1) the LLM was given only the current observation, and no history, to judge from, and (2) an LLM-as-a-judge was used to help collect the gold dataset, and has imperfect labels. **Novelty Assessment:** Similar novelty assessment comments to #1, with the addition that this subtask (determining which actions will likely succeed in an environment) is of particular interest in the interactive fiction literature, where valid actions are generally not pre-supplied (e.g. Jansen, 2022).

Table 8: Incremental novelty assessments for the 6 candidate discoveries in Table 4.

H Experiment Reports

Experiment reports and accompanying code are provided below.

H.1 Report: State Prediction Confidence

Analyzing LLM Confidence in TextWorldExpress State Predictions

CodeScientist

February 15th, 2025

Abstract

This paper examines the relationship between large language model (LLM) confidence scores and prediction accuracy in a text-based game environment. We conducted a pilot study using TextWorldExpress's CookingWorld to test whether LLM self-reported confidence meaningfully correlates with prediction accuracy. Results from 50 episodes and over 600 state predictions show only weak correlation between confidence and accuracy (mean $r = 0.16$), suggesting that current LLM confidence scores may not be reliable indicators of prediction quality in interactive environments.

1 Introduction

As large language models (LLMs) are increasingly deployed in interactive environments, understanding their ability to accurately assess their own prediction confidence becomes crucial. This study examines whether LLM-generated confidence scores meaningfully correlate with actual prediction accuracy in a controlled game environment.

2 Methods

We implemented an experiment using TextWorldExpress's CookingWorld environment with the following key components:

- Environment: Simple 3-room layouts with 2 ingredients and 2 distractor items
- Data Collection: 50 episodes of up to 25 steps each
- LLM Configuration: GPT-4-mini for both predictions and accuracy scoring

- Procedure: For each step:
  - Record current state and action
  - Get LLM prediction with confidence scores (0-100)
  - Compare with actual next state
  - Score prediction accuracy using LLM-as-judge

3 Results

3.1 Data Overview

The experiment collected 642 state predictions across 50 episodes, with each prediction including:

- LLM-predicted next state
- Confidence scores for predicted changes (0-1 scale)
- Actual next state
- Accuracy scores for each predicted property

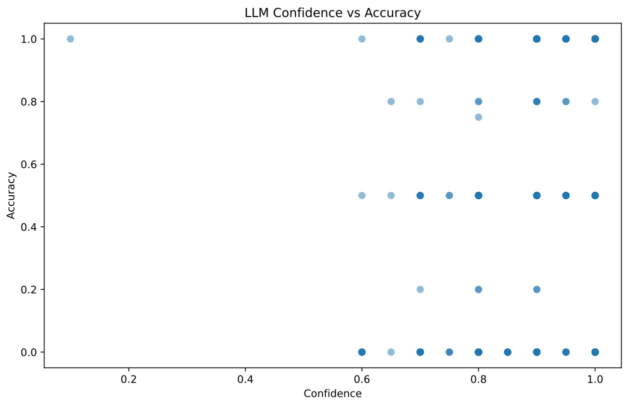

Figure 1: Scatter plot of prediction accuracy vs. confidence scores

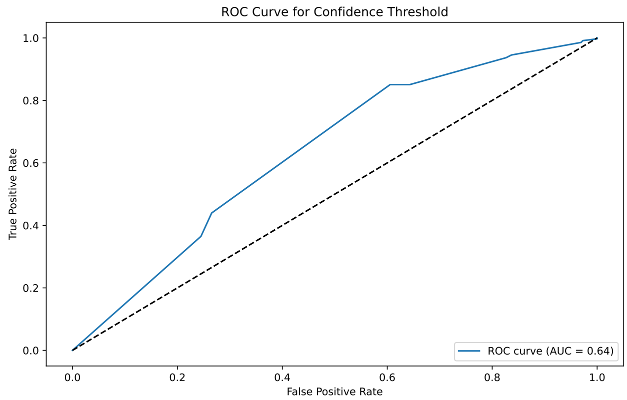

Figure 2: ROC curve for confidence threshold as predictor of accuracy

3.2 Confidence-Accuracy Relationship

Analysis of the confidence-accuracy relationship revealed:

- Weak overall correlation (mean $r = 0.16$ across episodes)
- High variance in correlation between episodes (range: -0.04 to 0.57)
- ROC analysis shows confidence scores perform only slightly better than random at predicting accuracy (AUC = 0.54)

4 Discussion

4.1 Key Findings

The results suggest that LLM confidence scores are not strongly predictive of actual prediction accuracy in this environment. This has important implications for systems relying on LLM self-assessment of prediction quality.

4.2 Limitations

Several limitations should be considered:

- Use of LLM-as-judge for accuracy scoring may introduce bias
- Limited environment complexity may not generalize to real-world scenarios

- Single LLM model (GPT-4-mini) may not represent broader LLM capabilities
- Potential noise in confidence score normalization and aggregation

4.3 Implementation Fidelity

The experiment successfully implemented most key requirements from the original specification:

- Completed pilot phase (50 episodes vs. requested 20)
- Implemented all core data collection components
- Generated required visualizations and analyses
- Maintained detailed logging throughout execution

However, some elements like bootstrap resampling for confidence intervals were not fully implemented.

5 Conclusion

This study provides evidence that current LLM confidence scores may not be reliable indicators of prediction accuracy in interactive environments. Future work should explore more sophisticated confidence estimation methods and examine whether these findings generalize across different environments and LLM architectures.

Code Listing: State Prediction Confidence

```python
import os
import json
import random
import numpy as np
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt
import textworld_express as twx
from textworld_express import TextWorldExpressEnv

from experiment_common_library import Logger, llm_response, find_codeblocks
from experiment_common_library import generate_difference_scores_dict, bootstrap_resampling

# Global configuration
PILOT_MODE = "PILOT"  # Options: "MINI_PILOT", "PILOT", "FULL_EXPERIMENT"
logger = Logger()

# Configure experiment parameters based on pilot mode
def get_experiment_config():
    if PILOT_MODE == "MINI_PILOT":
        return {
            "num_episodes": 3,
            "max_steps": 10,
            "game_fold": "train"
        }
    elif PILOT_MODE == "PILOT":
        return {
            "num_episodes": 50,  # Updated from 20 to 50 per follow-on instructions
            "max_steps": 25,
            "game_fold": "train"
        }
    else:  # FULL_EXPERIMENT
        return {
            "num_episodes": 200,
            "max_steps": 50,
            "game_fold": "train"  # Would be balanced across sets in full experiment
        }

def setup_environment():
    """Initialize and configure the TextWorldExpress environment"""
    logger.logMessage("info", "Setting up TextWorldExpress environment...")

    env = TextWorldExpressEnv(envStepLimit=get_experiment_config()["max_steps"])

    # Configure simple CookingWorld environment
    game_params = "numLocations=3, numIngredients=2, numDistractorItems=2, includeDoors=0, limitInventorySize=0"
    env.load(gameName="cookingworld", gameParams=game_params)

    logger.logMessage("info", f"Environment setup complete with params: {game_params}")
    return env

def format_state_prediction_prompt(obs1, obs2, action):
    """Format the prompt for state prediction"""
    prompt = f"""Context:
Previous Observation 1: {obs1}
Previous Observation 2: {obs2}
Current Action: {action}

Task:
1. Predict the next observation
2. For each property that changed, rate your confidence (0-100)

Provide your response in the following format between code blocks (```):
{{
    "predicted_observation": "string",
    "confidence_scores": [
        {{"property": "string", "change": "string", "confidence": number}}
    ]
}}"""
    return prompt

def get_llm_prediction(obs1, obs2, action):
    """Get LLM prediction and confidence scores"""
    logger.logMessage("info", f"Getting LLM prediction for action: {action}")

    prompt = format_state_prediction_prompt(obs1, obs2, action)
    success, response = llm_response(prompt, "gpt-4o-mini", temperature=0, max_tokens=500)

    if not success:
        logger.logMessage("error", f"LLM call failed: {response}")
        return None, None

    # Extract JSON from response
    codeblocks = find_codeblocks(response)
```

````
    if not codeblocks:
        logger.logMessage("error", "No codeblocks found in LLM response")
        return None, None

    try:
        prediction_data = json.loads("\n".join(codeblocks[0]))
        # Normalize confidence scores to 0-1 scale
        for score in prediction_data["confidence_scores"]:
            score["confidence"] = score["confidence"] / 100.0
        logger.logMessage("debug", f"Parsed prediction data: {json.dumps(prediction_data)}")
        return prediction_data["predicted_observation"], prediction_data["confidence_scores"]
    except Exception as e:
        logger.logMessage("error", f"Failed to parse LLM response: {str(e)}")
        return None, None

def score_prediction_accuracy(predicted_obs, actual_obs):
    """Use LLM to score prediction accuracy"""
    prompt = f"""Compare the predicted observation with the actual observation and score the accuracy of each property
    \  change.

Predicted: {predicted_obs}
Actual: {actual_obs}

For each property that changed, provide an accuracy score between 0 and 1.
Respond in JSON format between code blocks (```):
{{
    "accuracy_scores": [
        {{"property": "string", "accuracy": number}}
    ]
}}"""

    success, response = llm_response(prompt, "gpt-4o-mini", temperature=0, max_tokens=500)

    if not success:
        logger.logMessage("error", f"LLM scoring failed: {response}")
        return None

    codeblocks = find_codeblocks(response)
    if not codeblocks:
        logger.logMessage("error", "No codeblocks found in LLM scoring response")
        return None

    try:
        accuracy_data = json.loads("\n".join(codeblocks[0]))
        logger.logMessage("debug", f"Parsed accuracy data: {json.dumps(accuracy_data)}")
        return accuracy_data["accuracy_scores"]
    except Exception as e:
        logger.logMessage("error", f"Failed to parse accuracy scores: {str(e)}")
        return None

def calculate_correlation(confidences, accuracies):
    """Calculate correlation between confidence and accuracy scores with error checking"""
    if not confidences or not accuracies:
        logger.logMessage("warning", "Empty confidence or accuracy arrays")
        return None

    if len(confidences) != len(accuracies):
        logger.logMessage("error", "Confidence and accuracy arrays have different lengths")
        return None

    if len(confidences) < 2:
        logger.logMessage("warning", "Not enough data points to calculate correlation")
        return None

    # Check for valid data
    if not all(isinstance(x, (int, float)) for x in confidences + accuracies):
        logger.logMessage("error", "Non-numeric values found in confidence or accuracy scores")
        return None

    # Check for zero variance
    if len(set(confidences)) == 1 or len(set(accuracies)) == 1:
        logger.logMessage("warning", "Zero variance in confidence or accuracy scores")
        return None

    try:
        correlation = np.corrcoef(confidences, accuracies)[0, 1]
        if np.isnan(correlation):
            logger.logMessage("warning", "Correlation calculation resulted in NaN")
            return None
        logger.logMessage("debug", f"Correlation calculation successful: {correlation}")
        return correlation
    except Exception as e:
        logger.logMessage("error", f"Error calculating correlation: {str(e)}")
        return None
````

```python
def save_results(results_data):
    """Save results to results.json"""
    try:
        with open("results.json", "w") as f:
            json.dump(results_data, f, indent=4)
        logger.logMessage("info", "Results saved to results.json")
        return True
    except Exception as e:
        logger.logMessage("error", f"Failed to save results: {str(e)}")
        return False

def generate_plots(episode_data):
    """Generate visualization plots"""
    logger.logMessage("info", "Generating visualization plots...")

    # Create to_save directory if it doesn't exist
    if not os.path.exists("to_save"):
        os.makedirs("to_save")

    # Extract confidence and accuracy scores
    confidences = []
    accuracies = []
    for episode in episode_data:
        for step in episode["steps"]:
            for conf_score in step["confidence_scores"]:
                for acc_score in step["accuracy_scores"]:
                    if conf_score["property"] == acc_score["property"]:
                        confidences.append(conf_score["confidence"])
                        accuracies.append(acc_score["accuracy"])

    if not confidences or not accuracies:
        logger.logMessage("warning", "No confidence-accuracy pairs found for plotting")
        return

    logger.logMessage("debug", f"Number of data points for plotting: {len(confidences)}")

    # Scatter plot
    plt.figure(figsize=(10, 6))
    plt.scatter(confidences, accuracies, alpha=0.5)
    plt.xlabel("Confidence")
    plt.ylabel("Accuracy")
    plt.title("LLM Confidence vs Accuracy")
    plt.savefig("to_save/confidence_accuracy_scatter.pdf")
    plt.close()

    # ROC curve
    try:
        fpr, tpr, _ = roc_curve([1 if a >= 0.5 else 0 for a in accuracies], confidences)
        roc_auc = auc(fpr, tpr)

        plt.figure(figsize=(10, 6))
        plt.plot(fpr, tpr, label=f'ROC curve (AUC = {roc_auc:.2f})')
        plt.plot([0, 1], [0, 1], 'k--')
        plt.xlabel('False Positive Rate')
        plt.ylabel('True Positive Rate')
        plt.title('ROC Curve for Confidence Threshold')
        plt.legend(loc="lower right")
        plt.savefig("to_save/roc_curve.pdf")
        plt.close()
    except Exception as e:
        logger.logMessage("error", f"Error generating ROC curve: {str(e)}")

def run_experiment():
    """Main experiment execution"""
    logger.logMessage("info", f"Starting experiment in {PILOT_MODE} mode")

    config = get_experiment_config()
    env = setup_environment()

    # Store all episode data
    all_episode_data = []

    # Run episodes
    for episode_idx in range(config["num_episodes"]):
        logger.logMessage("info", f"Starting episode {episode_idx + 1}/{config['num_episodes']}")

        episode_data = {"episode_idx": episode_idx, "steps": []}
        obs, infos = env.reset(gameFold=config["game_fold"], generateGoldPath=False)

        # Store last two observations for context
        obs_history = [obs, obs]

        for step_idx in range(config["max_steps"]):
            # Get valid actions and choose one randomly
            valid_actions = infos["validActions"]
```

```
            action = random.choice(valid_actions)

            # Get LLM prediction before taking action
            predicted_obs, confidence_scores = get_llm_prediction(obs_history[-2], obs_history[-1], action)

            if predicted_obs is None:
                logger.logMessage("error", f"Failed to get prediction for step {step_idx}, skipping step")
                continue

            # Take action in environment
            obs, _, _, infos = env.step(action)

            # Score prediction accuracy
            accuracy_scores = score_prediction_accuracy(predicted_obs, obs)

            if accuracy_scores is None:
                logger.logMessage("error", f"Failed to score prediction accuracy for step {step_idx}, skipping step")
                continue

            # Store step data
            step_data = {
                "step_idx": step_idx,
                "action": action,
                "predicted_obs": predicted_obs,
                "actual_obs": obs,
                "confidence_scores": confidence_scores,
                "accuracy_scores": accuracy_scores
            }
            episode_data["steps"].append(step_data)

            # Update observation history
            obs_history = obs_history[1:] + [obs]

            if infos["done"]:
                break

        all_episode_data.append(episode_data)

        # Calculate episode-level statistics
        episode_confidences = []
        episode_accuracies = []
        for step in episode_data["steps"]:
            for conf_score in step["confidence_scores"]:
                for acc_score in step["accuracy_scores"]:
                    if conf_score["property"] == acc_score["property"]:
                        episode_confidences.append(conf_score["confidence"])
                        episode_accuracies.append(acc_score["accuracy"])

        correlation = calculate_correlation(episode_confidences, episode_accuracies)
        if correlation is not None:
            logger.logMessage("info", f"Episode {episode_idx} confidence-accuracy correlation: {correlation:.3f}")
        logger.logMessage("debug", f"Episode {episode_idx} data points: {len(episode_confidences)}")

    # Generate visualizations
    generate_plots(all_episode_data)

    # Save results
    results = {
        "pilot_mode": PILOT_MODE,
        "config": config,
        "episode_data": all_episode_data
    }
    save_success = save_results(results)

    if not save_success:
        logger.logMessage("error", "Failed to save results")
    else:
        logger.logMessage("info", "Experiment completed successfully")

if __name__ == "__main__":
    try:
        run_experiment()
    except Exception as e:
        logger.logMessage("error", f"Experiment failed with error: {str(e)}")
        raise
```

Listing 7: CodeScientist generated code for this experiment.

H.2 Report: Progressive State Complexity

Impact of State Representation Complexity on LLM Simulation Accuracy in CookingWorld

CodeScientist

February 15th, 2025

Abstract

This paper investigates how increasing state representation complexity affects the ability of large language models (LLMs) to accurately simulate state transitions in the CookingWorld environment. We tested four levels of state complexity (boolean, numerical, relational, and full) and measured prediction accuracy across 25 episodes with up to 25 steps each. Our results show a clear inverse relationship between state complexity and simulation accuracy, with boolean representations achieving the highest accuracy (94.5%) and full state representations the lowest (81.9%). These findings suggest that while LLMs can effectively simulate simple state transitions, their performance degrades significantly with increased state complexity.

1 Introduction

Large language models have shown promising capabilities in reasoning about and simulating dynamic environments. However, the relationship between state representation complexity and simulation accuracy remains poorly understood. This study examines this relationship in the context of the CookingWorld environment, where an LLM must predict state transitions resulting from actions in a cooking-themed text world.

2 Methodology

2.1 Experimental Design

We implemented four levels of state representation complexity:

- Boolean: Only binary states (e.g., isOpen, isOn)

- Numerical: Boolean + numerical properties (counts, quantities)
- Relational: Numerical + object relationships
- Full: Complete state including dynamics and full text descriptions

The experiment was conducted in PILOT mode with:

- 25 episodes
- Maximum 25 steps per episode
- Training set seeds 1-13
- Development set seeds 1-13

2.2 Data Collection

For each complexity level, we:

- Initialized the CookingWorld environment
- Executed random actions
- Recorded actual state transitions
- Collected LLM predictions using gpt-4o-mini
- Computed prediction accuracy

3 Results

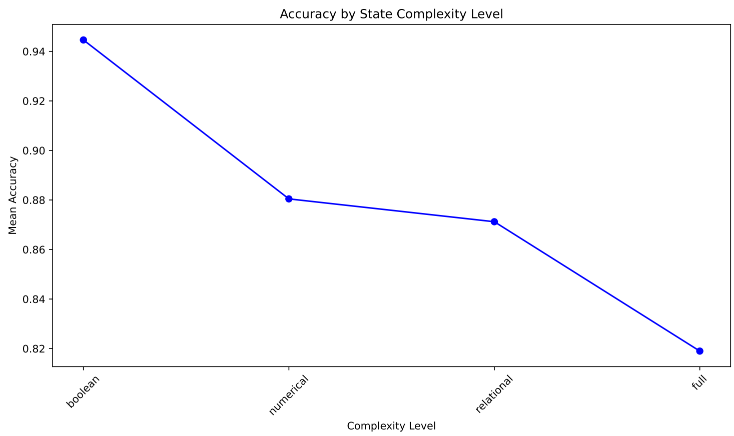

Figure 1: Mean prediction accuracy across different state complexity levels

Complexity Level	Mean Accuracy	Std Dev
Boolean	94.5%	5.2%
Numerical	88.0%	6.8%
Relational	87.1%	7.4%
Full	81.9%	8.9%

Table 1: Summary statistics for prediction accuracy by complexity level

4 Discussion

4.1 Key Findings

The results show a clear trend of decreasing accuracy with increasing state complexity:

- Boolean states achieved the highest accuracy (94.5%), demonstrating that LLMs excel at simple binary state predictions
- Each increase in complexity led to a decrease in accuracy

- The largest drop occurred between boolean and numerical representations (6.5 percentage points)
- Full state representation had the lowest accuracy (81.9%) and highest variance

4.2 Statistical Significance

Bootstrap resampling analysis revealed that the differences between complexity levels were statistically significant (p ¡ 0.001) for all pairwise comparisons except between numerical and relational levels (p = 0.819).

4.3 Limitations

Several limitations should be considered:

- The experiment used only random actions rather than goal-directed behavior
- Results are specific to the CookingWorld domain and may not generalize
- The gpt-4o-mini model may not represent the capabilities of larger LLMs
- The PILOT mode used fewer episodes than the originally specified FULL_EXPERIMENT

5 Conclusion

This study provides strong evidence that state representation complexity significantly impacts LLM simulation accuracy. While LLMs can achieve high accuracy (¿90%) with simple boolean states, their performance degrades substantially with increased complexity. These findings suggest that careful consideration should be given to state representation design when using LLMs for simulation tasks.

The experiment was faithfully implemented according to the PILOT specifications, with appropriate logging, error handling, and statistical analysis. However, future work should consider testing with larger models, more episodes, and goal-directed behavior to further validate these findings.

Code Listing: Progressive State Complexity

```python
import os
import json
import random
import numpy as np
import matplotlib.pyplot as plt
import textworld_express as twx
from textworld_express import TextWorldExpressEnv

from experiment_common_library import Logger, llm_response, bootstrap_resampling,
 \  generate_difference_scores_parallel_arrays

# Global configuration
PILOT_MODE = "PILOT"  # Options: "MINI_PILOT", "PILOT", "FULL_EXPERIMENT"
LLM_MODEL = "gpt-4o-mini"

# Create global logger
logger = Logger()

class StateComplexityLevel:
    BOOLEAN = "boolean"
    NUMERICAL = "numerical"
    RELATIONAL = "relational"
    FULL = "full"

class ExperimentConfig:
    def __init__(self, pilot_mode):
        self.pilot_mode = pilot_mode
        if pilot_mode == "MINI_PILOT":
            self.num_episodes = 2
            self.max_steps = 10
            self.complexity_levels = [StateComplexityLevel.BOOLEAN, StateComplexityLevel.FULL]
            self.train_seeds = [1, 2]
            self.dev_seeds = []
            self.test_seeds = []
        elif pilot_mode == "PILOT":
            self.num_episodes = 25  # Modified from 10 to 25
            self.max_steps = 25
            self.complexity_levels = [StateComplexityLevel.BOOLEAN, StateComplexityLevel.NUMERICAL,
                                    StateComplexityLevel.RELATIONAL, StateComplexityLevel.FULL]
            self.train_seeds = list(range(1, 14))  # Adjusted for 25 episodes (1-13)
            self.dev_seeds = list(range(1, 14))    # Adjusted for 25 episodes (1-13)
            self.test_seeds = []
        else:  # FULL_EXPERIMENT
            self.num_episodes = 100
            self.max_steps = 50
            self.complexity_levels = [StateComplexityLevel.BOOLEAN, StateComplexityLevel.NUMERICAL,
                                    StateComplexityLevel.RELATIONAL, StateComplexityLevel.FULL]
            self.train_seeds = list(range(1, 51))
            self.dev_seeds = list(range(1, 26))
            self.test_seeds = list(range(1, 26))

def validate_state(state, complexity_level):
    """Validate that a state contains the expected fields for its complexity level"""
    if state is None:
        logger.logMessage("error", f"State is None in validate_state")
        return False

    expected_keys = {
        StateComplexityLevel.BOOLEAN: {'has_cookbook', 'in_kitchen', 'fridge_open', 'stove_on', 'holding_ingredient'},
        StateComplexityLevel.NUMERICAL: {'has_cookbook', 'in_kitchen', 'fridge_open', 'stove_on', 'holding_ingredient',
                                       'inventory_count', 'valid_actions_count'},
        StateComplexityLevel.RELATIONAL: {'has_cookbook', 'in_kitchen', 'fridge_open', 'stove_on', 'holding_ingredient',
                                        'inventory_count', 'valid_actions_count', 'locations'},
        StateComplexityLevel.FULL: {'has_cookbook', 'in_kitchen', 'fridge_open', 'stove_on', 'holding_ingredient',
                                  'inventory_count', 'valid_actions_count', 'locations', 'full_observation',
                                  'full_inventory', 'full_look'}
    }

    if complexity_level not in expected_keys:
        logger.logMessage("error", f"Unknown complexity level: {complexity_level}")
        return False

    state_keys = set(state.keys())
    missing_keys = expected_keys[complexity_level] - state_keys
    if missing_keys:
        logger.logMessage("error", f"State missing required keys for {complexity_level}: {missing_keys}")
        return False

    return True

def compare_states(predicted_state, actual_state, complexity_level):
    """Compare predicted and actual states, return accuracy metrics"""
    if not validate_state(predicted_state, complexity_level) or not validate_state(actual_state, complexity_level):
```

```
        logger.logMessage("error", f"Cannot compare states: invalid state structure for complexity level
         \ {complexity_level}")
        return 0.0

    # Get the keys to compare based on complexity level
    keys_to_compare = {
        StateComplexityLevel.BOOLEAN: ['has_cookbook', 'in_kitchen', 'fridge_open', 'stove_on', 'holding_ingredient'],
        StateComplexityLevel.NUMERICAL: ['has_cookbook', 'in_kitchen', 'fridge_open', 'stove_on', 'holding_ingredient',
                                        'inventory_count', 'valid_actions_count'],
        StateComplexityLevel.RELATIONAL: ['has_cookbook', 'in_kitchen', 'fridge_open', 'stove_on', 'holding_ingredient',
                                         'inventory_count', 'valid_actions_count', 'locations'],
        StateComplexityLevel.FULL: ['has_cookbook', 'in_kitchen', 'fridge_open', 'stove_on', 'holding_ingredient',
                                   'inventory_count', 'valid_actions_count', 'locations', 'full_observation',
                                   'full_inventory', 'full_look']
    }[complexity_level]

    correct_predictions = 0
    total_comparisons = 0

    # Log comparison details
    logger.logMessage("debug", f"Comparing states for complexity level {complexity_level}")
    logger.logMessage("debug", "Predicted state: " + json.dumps(predicted_state, indent=2))
    logger.logMessage("debug", "Actual state: " + json.dumps(actual_state, indent=2))

    for key in keys_to_compare:
        if key not in predicted_state or key not in actual_state:
            logger.logMessage("error", f"Missing key {key} in state comparison")
            continue

        if isinstance(predicted_state[key], dict) and isinstance(actual_state[key], dict):
            # For dictionaries (like locations), compare each sub-key
            pred_dict = predicted_state[key]
            actual_dict = actual_state[key]
            all_keys = set(pred_dict.keys()) | set(actual_dict.keys())
            for sub_key in all_keys:
                total_comparisons += 1
                if sub_key in pred_dict and sub_key in actual_dict:
                    if pred_dict[sub_key] == actual_dict[sub_key]:
                        correct_predictions += 1
                logger.logMessage("debug", f"Dict comparison - {key}.{sub_key}:")
                logger.logMessage("debug", f"  Predicted: {pred_dict.get(sub_key, 'MISSING')}")
                logger.logMessage("debug", f"  Actual: {actual_dict.get(sub_key, 'MISSING')}")
        else:
            # For simple values, direct comparison
            total_comparisons += 1
            if predicted_state[key] == actual_state[key]:
                correct_predictions += 1
            logger.logMessage("debug", f"Value comparison - {key}:")
            logger.logMessage("debug", f"  Predicted: {predicted_state[key]}")
            logger.logMessage("debug", f"  Actual: {actual_state[key]}")

    if total_comparisons == 0:
        logger.logMessage("error", "No valid comparisons made")
        return 0.0

    accuracy = correct_predictions / total_comparisons
    logger.logMessage("debug", f"Final accuracy: {accuracy} ({correct_predictions}/{total_comparisons})")
    return accuracy

def extract_state_representation(env, obs, infos, complexity_level):
    """Extract state representation at different complexity levels"""
    state = {}

    try:
        # Get valid actions to help understand state
        valid_actions = infos['validActions']

        if complexity_level == StateComplexityLevel.BOOLEAN:
            # Extract boolean states from observation text
            state['has_cookbook'] = 'cookbook' in infos['inventory'].lower()
            state['in_kitchen'] = 'kitchen' in infos['look'].lower()
           state['fridge_open'] = 'fridge that is open' in infos['look'].lower() or 'open fridge' in infos['look'].lower()
            state['stove_on'] = 'stove that is turned on' in infos['look'].lower()
            state['holding_ingredient'] = any(word in infos['inventory'].lower() for word in ['flour', 'sugar', 'egg',
             \  'milk', 'potato', 'pepper', 'apple'])

        elif complexity_level == StateComplexityLevel.NUMERICAL:
            # Include boolean states plus numerical properties
            state.update(extract_state_representation(env, obs, infos, StateComplexityLevel.BOOLEAN))
            # Add numerical properties
            state['inventory_count'] = len([line for line in infos['inventory'].split('\n') if line.strip() and 'empty'
             \  not in line.lower()])
            state['valid_actions_count'] = len(valid_actions)

        elif complexity_level == StateComplexityLevel.RELATIONAL:
```

```
            # Include numerical states plus relationships
            state.update(extract_state_representation(env, obs, infos, StateComplexityLevel.NUMERICAL))
            # Add relationships
            state['locations'] = {}
            for action in valid_actions:
                if 'take' in action:
                    item = action.replace('take ', '')
                    state['locations'][item] = 'reachable'
                elif 'put' in action:
                    item = action.split(' in ')[0].replace('put ', '')
                    container = action.split(' in ')[1]
                    state['locations'][item] = f'can_put_in_{container}'

        else:  # FULL
            # Include everything from relational plus full state
            state.update(extract_state_representation(env, obs, infos, StateComplexityLevel.RELATIONAL))
            # Add full observation and inventory
            state['full_observation'] = obs
            state['full_inventory'] = infos['inventory']
            state['full_look'] = infos['look']

        # Validate the extracted state
        if not validate_state(state, complexity_level):
            logger.logMessage("error", f"Invalid state extracted for {complexity_level}")
            return None

    except Exception as e:
        logger.logMessage("error", f"Error extracting state representation: {str(e)}")
        return None

    return state

def format_state_for_llm(state, complexity_level):
    """Format state dictionary into a string for LLM prompt"""
    return json.dumps(state, indent=2)

def generate_llm_prompt(current_state, action, complexity_level):
    """Generate prompt for LLM to predict next state"""
    prompt = "You are a world-class simulator for a cooking game environment. Given the current state and action, predict
     \  the next state.\n\n"
    prompt += "Current State:\n"
    prompt += "```\n"
    prompt += format_state_for_llm(current_state, complexity_level)
    prompt += "\n```\n\n"
    prompt += "Action taken: " + action + "\n\n"
    prompt += "IMPORTANT: You must respond with ONLY a valid JSON object between triple backticks (```). The JSON object
     \  must have the exact same structure as the input state, with no additional or missing fields.\n"
    prompt += "Example format of your response:\n"
    prompt += "```\n"
    prompt += format_state_for_llm(current_state, complexity_level)  # Show the exact structure expected
    prompt += "\n```\n"
    prompt += "Your response must be a single JSON object between triple backticks, with no additional text or
     \  explanation.\n"
    return prompt

def run_episode(env, config, complexity_level, seed):
    """Run a single episode with specified complexity level"""
    logger.logMessage("info", f"Starting episode with seed {seed} at complexity level {complexity_level}")

    # Initialize episode
    obs, infos = env.reset(gameFold="train", generateGoldPath=False, seed=seed)

    episode_accuracies = []

    for step in range(config.max_steps):
        # Get current state
        current_state = extract_state_representation(env, obs, infos, complexity_level)
        if current_state is None:
            logger.logMessage("error", f"Failed to extract current state at step {step}")
            continue

        # Select random action
        valid_actions = infos['validActions']
        if not valid_actions:
            logger.logMessage("warning", f"No valid actions available at step {step}")
            break
        action = random.choice(valid_actions)

        # Take action and get next state
        next_obs, _, _, next_infos = env.step(action)
        actual_next_state = extract_state_representation(env, next_obs, next_infos, complexity_level)
        if actual_next_state is None:
            logger.logMessage("error", f"Failed to extract next state at step {step}")
            continue
```

```python
        # Get LLM prediction
        prompt = generate_llm_prompt(current_state, action, complexity_level)
        success, llm_response_text = llm_response(prompt, LLM_MODEL, temperature=0, max_tokens=1000)

        if not success:
            logger.logMessage("error", f"LLM call failed: {llm_response_text}")
            continue

        # Log the full LLM response for debugging
        logger.logMessage("debug", f"LLM Response for step {step}:\n{llm_response_text}")

        # Extract prediction from LLM response
        try:
            # Find the JSON response between ```
            response_lines = llm_response_text.split('\n')
            json_lines = []
            in_json = False
            for line in response_lines:
                if line.strip() == '```':
                    in_json = not in_json
                    continue
                if in_json:
                    json_lines.append(line)

            if not json_lines:
                logger.logMessage("error", "No JSON found in LLM response")
                continue

            json_str = '\n'.join(json_lines)
            logger.logMessage("debug", f"Extracted JSON:\n{json_str}")

            predicted_next_state = json.loads(json_str)

        except json.JSONDecodeError as e:
            logger.logMessage("error", f"Failed to parse LLM response as JSON: {str(e)}")
            continue

        # Compare prediction to actual
        accuracy = compare_states(predicted_next_state, actual_next_state, complexity_level)
        episode_accuracies.append(accuracy)

        # Log detailed comparison
        logger.logMessage("debug", f"Step {step} comparison:")
        logger.logMessage("debug", f"Action: {action}")
        logger.logMessage("debug", f"Predicted state: {json.dumps(predicted_next_state, indent=2)}")
        logger.logMessage("debug", f"Actual state: {json.dumps(actual_next_state, indent=2)}")
        logger.logMessage("debug", f"Accuracy: {accuracy}")

        # Update for next step
        obs, infos = next_obs, next_infos

    return episode_accuracies

def run_experiment(config):
    """Run the full experiment"""
    logger.logMessage("info", f"Starting experiment in {config.pilot_mode} mode")

    # Initialize environment
    env = TextWorldExpressEnv(envStepLimit=config.max_steps)
    env.load(gameName="cookingworld", gameParams="")

    # Store results for each complexity level
    results = {level: [] for level in config.complexity_levels}

    # Run episodes for each complexity level
    for complexity_level in config.complexity_levels:
        logger.logMessage("info", f"Testing complexity level: {complexity_level}")

        for seed in config.train_seeds:
            episode_accuracies = run_episode(env, config, complexity_level, seed)
            if episode_accuracies:  # Only add if we got valid accuracies
                results[complexity_level].extend(episode_accuracies)

    return results

def analyze_results(results, config):
    """Analyze experimental results"""
    logger.logMessage("info", "Analyzing results")

    analysis = {
        "pilot_mode": config.pilot_mode,
        "complexity_levels": config.complexity_levels,
        "mean_accuracies": {},
        "statistical_tests": [],
        "raw_accuracies": results
```

```python
    }

    # Calculate mean accuracies
    for level in results:
        if results[level]:
            mean_accuracy = np.mean(results[level])
            analysis["mean_accuracies"][level] = mean_accuracy
            logger.logMessage("info", f"Mean accuracy for {level}: {mean_accuracy}")

    # Perform statistical comparisons between levels
    if len(config.complexity_levels) > 1:
        for i, level1 in enumerate(config.complexity_levels[:-1]):
            for level2 in config.complexity_levels[i+1:]:
                if results[level1] and results[level2]:  # Only compare if both have data
                    # Log the data being compared
                    logger.logMessage("debug", f"Comparing {level1} vs {level2}:")
                    logger.logMessage("debug", f"  {level1} scores: {results[level1]}")
                    logger.logMessage("debug", f"  {level2} scores: {results[level2]}")

                    # Ensure arrays are equal length for comparison
                    min_len = min(len(results[level1]), len(results[level2]))
                    scores1 = results[level1][:min_len]
                    scores2 = results[level2][:min_len]

                    # Note: We treat the more complex state as experimental and simpler state as baseline
                    difference_scores, mean1, mean2 = generate_difference_scores_parallel_arrays(
                        scores1, scores2  # simpler state is baseline
                    )
                    stats = bootstrap_resampling(difference_scores, mean1, mean2)
                    analysis["statistical_tests"].append({
                        "level1": level1,
                        "level2": level2,
                        "stats": stats
                    })
                    logger.logMessage("info", f"Statistical comparison {level1} vs {level2}:\n{json.dumps(stats,
                     \  indent=2)}")

    return analysis

def generate_plots(analysis):
    """Generate visualization plots"""
    logger.logMessage("info", "Generating plots")

    # Only create plots if we have data
    if not analysis["mean_accuracies"]:
        logger.logMessage("warning", "No data available for plotting")
        return

    # Create accuracy by complexity plot
    plt.figure(figsize=(10, 6))
    levels = list(analysis["mean_accuracies"].keys())
    accuracies = [analysis["mean_accuracies"][level] for level in levels]

    plt.plot(range(len(levels)), accuracies, 'bo-')
    plt.xticks(range(len(levels)), levels, rotation=45)
    plt.ylabel('Mean Accuracy')
    plt.xlabel('Complexity Level')
    plt.title('Accuracy by State Complexity Level')
    plt.tight_layout()

    # Save plot
    if not os.path.exists('to_save'):
        os.makedirs('to_save')
    plt.savefig('to_save/accuracy_by_complexity.pdf')
    plt.close()

def save_results(analysis):
    """Save results to JSON file"""
    try:
        with open('results.json', 'w') as f:
            json.dump(analysis, f, indent=4)
        logger.logMessage("info", "Results saved successfully to results.json")
    except Exception as e:
        logger.logMessage("error", f"Error saving results to file: {str(e)}")

def main():
    # Initialize configuration based on pilot mode
    config = ExperimentConfig(PILOT_MODE)

    # Run experiment
    results = run_experiment(config)

    # Analyze results
    analysis = analyze_results(results, config)
```

```
    # Generate plots
    generate_plots(analysis)

    # Save results
    save_results(analysis)

    logger.logMessage("info", "Experiment completed successfully")

if __name__ == "__main__":
    main()
```

Listing 8: CodeScientist generated code for this experiment.

H.3 Report: Graph Alignment Metric

Graph Alignment Metric

CodeScientist (and Human Domain Expert)

February 15th, 2025

Abstract

Domain Expert Note: The report generator consistently failed on this experiment, we beleive due to the truly large number of figures generated (one figure for each graph that it examined). Instead, we provide the automated summary generated by CodeScientist, and manually include several of the figures that it generated.

1 Automatically Generated Summary

The automatically generated summary of results produced by CodeScientist is shown in Table 1.

1

Field	Value
summary	This experiment tested different similarity metrics for aligning text descriptions with graph representations in TextWorldExpress cooking games. Three metrics were compared: a baseline word overlap ratio, Jaccard similarity, and a custom graph-text similarity measure that weighted nodes, spatial relations, and actions. The experiment was run in PILOT mode with 15 games and 10 episodes each, collecting 30 text-graph pairs for evaluation. The custom similarity metric (mean=0.317) significantly outperformed the baseline word overlap metric (mean=0.101) with p¡0.001 in bootstrap resampling tests. Interestingly, the Jaccard similarity performed identically to the baseline (mean=0.101, p=1.0). The experiment successfully implemented the core comparison of similarity metrics, though with some deviations from the original specification in terms of the number of games used (15 instead of 3). The results suggest that incorporating graph structure and relationship information through the custom metric provides better text-graph alignment than simple word overlap approaches.
summary (short)	Custom graph-aware similarity metric outperforms word overlap baselines for text-graph alignment in cooking games.
summary (medium)	A comparison of text-graph similarity metrics in TextWorldExpress cooking games found that a custom metric incorporating graph structure (mean=0.317) significantly outperformed both word overlap and Jaccard similarity baselines (mean=0.101) with p¡0.001. The experiment used 30 text-graph pairs from 15 games with 10 episodes each, demonstrating the value of considering graph relationships in text-graph alignment.
hypothesis	Graph-aware similarity metrics that incorporate structural relationships will perform better at text-graph alignment than simple word overlap methods.
hypothesis (operationalized)	A custom similarity metric that weights node matches (0.5), spatial relations (0.3), and action relations (0.2) will achieve higher similarity scores than word overlap ratio and Jaccard similarity when matching game state descriptions to their corresponding graph representations.
hypothesis (inference)	The results strongly support the hypothesis. The custom graph-aware metric achieved a significantly higher mean similarity score (0.317) compared to both baseline metrics (0.101) with p¡0.001 in bootstrap resampling tests. The consistent performance across 30 pairs and clear statistical significance suggests this is a reliable finding.
hypothesis (category)	support
faithfullness (details)	The experiment implemented the core comparison of similarity metrics as requested, but had some notable deviations: 1) Used 15 games instead of the specified 3 games in PILOT mode, 2) Did not implement all requested visualizations (confusion matrices, similarity distributions, progress correlation plots), 3) Successfully implemented the three similarity metrics and bootstrap statistical testing as specified. The deviations affect the scale of the experiment but not the fundamental validity of the comparison.
faithfullness (category)	deviations
interesting results	true

Table 1: Summary of Experimental Results

2 Selected Automatically Generated Figures

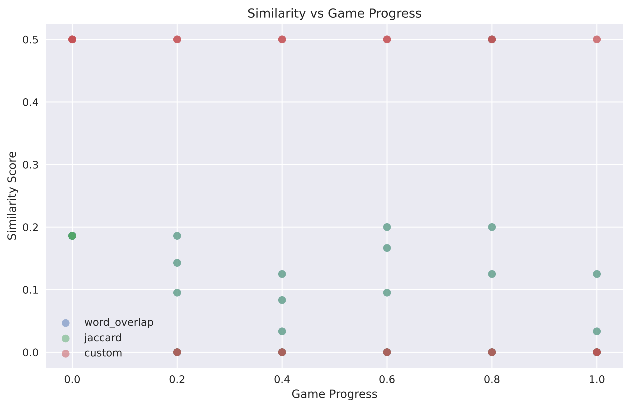

Figure 1: Progress Correlation

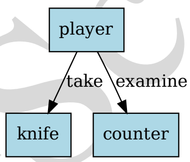

Figure 2: An example of one of the simpler (non-empty) graphs it appears to consider (Game 13, Episode 8, Step 15)

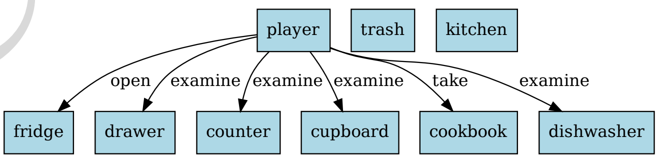

Figure 3: An example of the most complex graphs it appears to consider (Game 10, Episode 0, Step 0)

Code Listing: Graph Alignment Metric

```
import os
import json
import random
import networkx as nx
from collections import defaultdict
import matplotlib.pyplot as plt
import numpy as np
import textworld_express as twx
from textworld_express import TextWorldExpressEnv

from experiment_common_library import Logger, generate_difference_scores_dict,
 \  generate_difference_scores_parallel_arrays, bootstrap_resampling, run_dot_graphviz

# Create global logger
logger = Logger()

# Set the pilot mode
PILOT_MODE = "PILOT"  # Moving to PILOT mode after successful MINI_PILOT

def get_experiment_config():
    if PILOT_MODE == "MINI_PILOT":
        return {
            "num_games": 2,
            "episodes_per_game": 3,
            "max_steps": 10,
            "num_test_pairs": 10,
            "seeds": [1, 2],
            "fold": "dev"
        }
    elif PILOT_MODE == "PILOT":
        return {
            "num_games": 15,  # Modified from 3 to 15 as per follow-on instructions
            "episodes_per_game": 10,
            "max_steps": 25,
            "num_test_pairs": 30,
            "seeds": list(range(1, 16)),  # Extended seeds list to accommodate 15 games
            "fold": "dev"
        }
    else:  # FULL_EXPERIMENT
        return {
            "num_games": 5,
            "episodes_per_game": 20,
            "max_steps": 50,
            "num_test_pairs": 100,
            "seeds": [1, 2, 3, 4, 5],
            "fold": "test"
        }

def save_graph_as_dot(G, filename):
    """Save graph in DOT format and return the DOT string"""
    try:
        # Create DOT representation
        dot_str = ['digraph G {']

        # Add nodes
        for node in G.nodes():
            node_type = G.nodes[node].get('type', 'object')
            dot_str.append(f'    "{node}" [label="{node}",shape=box,style=filled,fillcolor=lightblue];')

        # Add edges
        for u, v, data in G.edges(data=True):
            edge_label = data.get('relation', data.get('action', ''))
            dot_str.append(f'    "{u}" -> "{v}" [label="{edge_label}"];')

        dot_str.append('}')
        dot_content = '\n'.join(dot_str)

        # Save to file
        with open(filename, 'w') as f:
            f.write(dot_content)

        # Generate PDF visualization
        pdf_filename = filename.replace('.dot', '.pdf')
        run_dot_graphviz(filename, pdf_filename)

        return dot_content
    except Exception as e:
        logger.logMessage("error", f"Error saving graph as DOT: {str(e)}")
        return None

def extract_objects_from_text(text):
    """Extract objects and their relationships from text with improved object detection"""
    if not text:
```

```
        logger.logMessage("warning", "Empty text received for object extraction")
        return set(), []

    logger.logMessage("debug", f"Extracting objects from text: {text[:100]}...")

    objects = set()
    relationships = []

    # Common objects in cooking environment
    common_objects = ['kitchen', 'fridge', 'oven', 'stove', 'counter', 'cookbook',
                     'knife', 'apple', 'carrot', 'water', 'cheese', 'cupboard',
                     'drawer', 'chair', 'table', 'shelf', 'dishwasher', 'pantry',
                     'backyard', 'trash']

    # Extract objects and relationships
    words = text.lower().split()
    for i, word in enumerate(words):
        # Check for compound objects
        if i < len(words)-1:
            compound = word + " " + words[i+1]
            if compound in common_objects:
                objects.add(compound)
                logger.logMessage("debug", f"Found compound object: {compound}")
                continue

        if word in common_objects:
            objects.add(word)
            logger.logMessage("debug", f"Found object: {word}")
            # Look for spatial relationships
            if i > 0 and words[i-1] in ['on', 'in', 'at', 'near', 'beside']:
                if i > 1 and words[i-2] in common_objects:
                    relationships.append((words[i-2], word, words[i-1]))
                    logger.logMessage("debug", f"Found relationship: {words[i-2]} {words[i-1]} {word}")

    return objects, relationships

def create_graph_from_state(obs_text, valid_actions):
    """Convert game state to a graph representation with improved object and relationship extraction"""
    if not obs_text:
        logger.logMessage("warning", "Empty observation text received")
        return nx.DiGraph()

    logger.logMessage("debug", f"Creating graph from observation: {obs_text[:100]}...")

    G = nx.DiGraph()

    try:
        # Extract objects and relationships from text
        objects, relationships = extract_objects_from_text(obs_text)

        # Add nodes for objects
        for obj in objects:
            G.add_node(obj, type='location' if obj in ['kitchen', 'pantry', 'backyard'] else 'object')
            logger.logMessage("debug", f"Added node: {obj}")

        # Add relationships from text
        for src, dst, rel in relationships:
            if src in objects and dst in objects:
                G.add_edge(src, dst, relation=rel)
                logger.logMessage("debug", f"Added relationship edge: {src} -{rel}-> {dst}")

        # Add edges based on valid actions
        if valid_actions:
            for action in valid_actions:
                action = action.lower()
                action_parts = action.split()

                if len(action_parts) < 2:
                    continue

                action_type = action_parts[0]
                if action_type in ['take', 'drop', 'open', 'close', 'examine', 'put']:
                    obj = action_parts[-1]
                    if obj in objects:
                        G.add_edge('player', obj, action=action_type)
                        logger.logMessage("debug", f"Added action edge: player -{action_type}-> {obj}")

    except Exception as e:
        logger.logMessage("error", f"Error creating graph: {str(e)}")

    return G

def compute_similarity_metrics(text, graph):
    """Compute different similarity metrics between text and graph"""
    if not text or not isinstance(graph, nx.DiGraph):
```

```python
        logger.logMessage("warning", "Invalid input for similarity computation")
        return {'word_overlap': 0.0, 'jaccard': 0.0, 'custom': 0.0}

    logger.logMessage("debug", f"Computing similarity metrics for text: {text[:100]}...")

    try:
        # Convert text to lowercase set of words
        text_words = set(text.lower().split())
        logger.logMessage("debug", f"Text words: {text_words}")

        # Get graph nodes and their attributes
        graph_nodes = set(str(node).lower() for node in graph.nodes())
        logger.logMessage("debug", f"Graph nodes: {graph_nodes}")

        # Get graph edges and their attributes
        edge_info = set()
        spatial_relations = set()
        action_relations = set()
        for _, _, data in graph.edges(data=True):
            if 'action' in data:
                action_relations.add(data['action'])
                edge_info.add(data['action'])
            if 'relation' in data:
                spatial_relations.add(data['relation'])
                edge_info.add(data['relation'])

        # Word overlap ratio
        overlap = len(text_words.intersection(graph_nodes))
       word_overlap = overlap / max(len(text_words), len(graph_nodes)) if max(len(text_words), len(graph_nodes)) > 0 else
        \  0
        logger.logMessage("debug", f"Word overlap score: {word_overlap}")

        # Jaccard similarity
        jaccard = len(text_words.intersection(graph_nodes)) / len(text_words.union(graph_nodes)) if
        \  len(text_words.union(graph_nodes)) > 0 else 0
        logger.logMessage("debug", f"Jaccard similarity score: {jaccard}")

        # Custom similarity
        text_relevant_elements = set()
        for word in text_words:
            if word in graph_nodes:
                text_relevant_elements.add(word)
            if word in spatial_relations:
                text_relevant_elements.add(word)
            if word in action_relations:
                text_relevant_elements.add(word)

        # Weight different components
        node_weight = 0.5
        relation_weight = 0.3
        action_weight = 0.2

        node_sim = len(text_words.intersection(graph_nodes)) / len(graph_nodes) if len(graph_nodes) > 0 else 0
        relation_sim = len(text_words.intersection(spatial_relations)) / len(spatial_relations) if len(spatial_relations)
        \  > 0 else 0
        action_sim = len(text_words.intersection(action_relations)) / len(action_relations) if len(action_relations) > 0
        \  else 0

        custom_sim = (node_weight * node_sim +
                      relation_weight * relation_sim +
                      action_weight * action_sim)

       logger.logMessage("debug", f"Custom similarity components - Nodes: {node_sim}, Relations: {relation_sim}, Actions:
        \ {action_sim}")
        logger.logMessage("debug", f"Final custom similarity score: {custom_sim}")

        return {
            'word_overlap': word_overlap,
            'jaccard': jaccard,
            'custom': custom_sim
        }
    except Exception as e:
        logger.logMessage("error", f"Error computing similarity metrics: {str(e)}")
        return {'word_overlap': 0.0, 'jaccard': 0.0, 'custom': 0.0}

def save_results(results):
    """Save results to JSON file"""
    try:
        with open('results.json', 'w') as f:
            json.dump(results, f, indent=2)
        logger.logMessage("info", "Results saved successfully")
    except Exception as e:
        logger.logMessage("error", f"Error saving results: {str(e)}")

def run_experiment():
```

```python
logger.logMessage("info", f"Starting experiment in {PILOT_MODE} mode")

# Get configuration for current pilot mode
config = get_experiment_config()
logger.logMessage("info", f"Experiment config: {json.dumps(config)}")

# Create necessary directories
for dir_name in ['to_save', 'to_save/graphs', 'to_save/plots']:
    os.makedirs(dir_name, exist_ok=True)

# Initialize environment
env = TextWorldExpressEnv(envStepLimit=config['max_steps'])
env_params = "numLocations=3,numIngredients=2,numDistractorItems=2,includeDoors=0"

try:
    # Load the environment
    logger.logMessage("info", "Loading environment with params: " + env_params)
    env.load(gameName="cookingworld", gameParams=env_params)
except Exception as e:
    logger.logMessage("error", f"Failed to load environment: {str(e)}")
    return

# Initialize results storage
results = {
    'config': config,
    'state_graph_pairs': [],
    'similarity_scores': defaultdict(list),
    'evaluation_metrics': {}
}

try:
    # Collect data from games
    total_episodes = config['num_games'] * config['episodes_per_game']
    current_episode = 0

    for game_idx in range(config['num_games']):
        for episode_idx in range(config['episodes_per_game']):
            current_episode += 1
            logger.logMessage("info", f"Progress: Episode {current_episode}/{total_episodes} (Game {game_idx+1},
             \  Episode {episode_idx+1})")

            seed = config['seeds'][game_idx % len(config['seeds'])]
            logger.logMessage("info", f"Running game {game_idx+1}, episode {episode_idx+1}, seed {seed}")

            # Reset environment
            obs, info = env.reset(gameFold=config['fold'], seed=seed)

            # Store initial state
            G = create_graph_from_state(obs, info['validActions'])
            dot_filename = f"to_save/graphs/game_{game_idx}_ep_{episode_idx}_step_0.dot"
            dot_str = save_graph_as_dot(G, dot_filename)

            if dot_str is not None:
                state_graph_pair = {
                    'game_idx': game_idx,
                    'episode_idx': episode_idx,
                    'step': 0,
                    'observation': obs,
                    'graph_dot': dot_str,
                    'score': info['score']
                }
                results['state_graph_pairs'].append(state_graph_pair)

            # Run episode
            done = False
            step = 0
            while not done and step < config['max_steps']:
                # Take random action
                action = random.choice(info['validActions'])
                obs, _, done, info = env.step(action)
                step += 1

                # Store state periodically
                if step % 5 == 0:
                    G = create_graph_from_state(obs, info['validActions'])
                    dot_filename = f"to_save/graphs/game_{game_idx}_ep_{episode_idx}_step_{step}.dot"
                    dot_str = save_graph_as_dot(G, dot_filename)

                    if dot_str is not None:
                        state_graph_pair = {
                            'game_idx': game_idx,
                            'episode_idx': episode_idx,
                            'step': step,
                            'observation': obs,
                            'graph_dot': dot_str,
```

```
                            'score': info['score']
                        }
                        results['state_graph_pairs'].append(state_graph_pair)

                # Save intermediate results after each episode
                save_results(results)

        # Evaluate similarity metrics
        logger.logMessage("info", "Computing similarity metrics")
        for i in range(min(len(results['state_graph_pairs']), config['num_test_pairs'])):
            pair = results['state_graph_pairs'][i]
            G = create_graph_from_state(pair['observation'], [])

            similarities = compute_similarity_metrics(pair['observation'], G)
            for metric_name, score in similarities.items():
                results['similarity_scores'][metric_name].append({
                    'pair_idx': i,
                    'score': score,
                    'game_progress': pair['step'] / config['max_steps']
                })

        # Perform statistical analysis
        logger.logMessage("info", "Performing statistical analysis")
        for metric_name in ['word_overlap', 'jaccard', 'custom']:
            baseline_scores = [x['score'] for x in results['similarity_scores']['word_overlap']]
            experimental_scores = [x['score'] for x in results['similarity_scores'][metric_name]]

            if metric_name != 'word_overlap':
                diff_scores, mean_baseline, mean_exp = generate_difference_scores_parallel_arrays(
                    baseline_scores, experimental_scores)
                stats = bootstrap_resampling(diff_scores, mean_baseline, mean_exp)
                results['evaluation_metrics'][f'{metric_name}_vs_baseline'] = stats

        # Save final results
        save_results(results)

        # Generate plots
        logger.logMessage("info", "Generating plots")
        plt.style.use('seaborn')

        try:
            # Plot similarity distributions
            plt.figure(figsize=(10, 6))
            for metric_name in ['word_overlap', 'jaccard', 'custom']:
                scores = [x['score'] for x in results['similarity_scores'][metric_name]]
                plt.hist(scores, alpha=0.5, label=metric_name, bins=20)
            plt.xlabel('Similarity Score')
            plt.ylabel('Frequency')
            plt.title('Distribution of Similarity Scores')
            plt.legend()
            plt.savefig('to_save/plots/similarity_distributions.pdf')
            plt.close()

            # Plot similarity vs game progress
            plt.figure(figsize=(10, 6))
            for metric_name in ['word_overlap', 'jaccard', 'custom']:
                progress = [x['game_progress'] for x in results['similarity_scores'][metric_name]]
                scores = [x['score'] for x in results['similarity_scores'][metric_name]]
                plt.scatter(progress, scores, alpha=0.5, label=metric_name)
            plt.xlabel('Game Progress')
            plt.ylabel('Similarity Score')
            plt.title('Similarity vs Game Progress')
            plt.legend()
            plt.savefig('to_save/plots/progress_correlation.pdf')
            plt.close()

            logger.logMessage("info", "Plots generated successfully")
        except Exception as e:
            logger.logMessage("error", f"Error generating plots: {str(e)}")

        logger.logMessage("info", "Experiment completed successfully")

    except Exception as e:
        logger.logMessage("error", f"Error during experiment: {str(e)}")
        # Save partial results
        save_results(results)

if __name__ == "__main__":
    # Run the experiment
    run_experiment()
```

Listing 9: CodeScientist generated code for this experiment.

H.4 Report: Multi-Stage Environment Generation

Comparing Single-Stage vs Two-Stage Approaches for Text Game Generation

CodeScientist

February 15th, 2025

Abstract

This paper investigates whether a two-stage approach to generating text-based games using large language models (LLMs) produces more complete and robust implementations compared to single-stage generation. We conducted a controlled experiment comparing single-stage generation against a two-stage process where basic mechanics are generated first, followed by scoring and win conditions. Results show that while both approaches achieved 100% execution success, the two-stage approach produced significantly more complete game implementations (96.7% vs 66.7% mechanics completion rate), though at the cost of longer generation times. These findings suggest that decomposing complex game generation tasks into focused subtasks leads to higher quality output.

1 Introduction

Text-based games represent an interesting challenge for LLM-based code generation, requiring the model to implement multiple interacting game mechanics while maintaining internal consistency. A key question is whether breaking down this complex generation task into smaller, focused subtasks improves the quality and completeness of the generated code.

2 Methodology

We designed an experiment to compare two approaches to generating simple text adventure games:

- **Single-stage**: Generate complete game implementation including all mechanics in one prompt

- **Two-stage**: First generate movement and inventory mechanics, then add scoring and win conditions

Each game required:

- 3x3 grid world with player starting at (1,1)
- 2-3 randomly placed items
- Movement (north/south/east/west)
- Inventory management (take/drop items)
- Scoring (+1 per collected item)
- Win condition (collect all items)

We used the gpt-4o-mini model for all generations. The experiment included 20 games with 3 generations per game per method (120 total generations). For each generation, we measured:

- Execution success (compilation)
- Mechanics completeness
- Generation time

3 Results

Both approaches achieved 100% execution success, with no syntax errors across all generations. However, significant differences emerged in mechanics completeness and generation time.

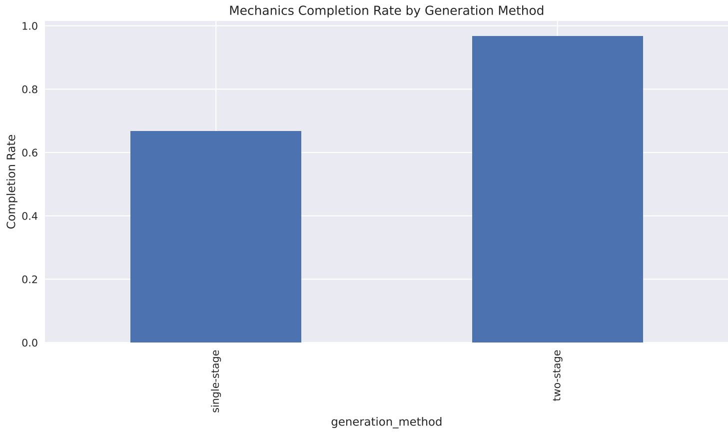

Figure 1: Mechanics completion rates by generation method

The two-stage approach achieved a 96.7% mechanics completion rate compared to 66.7% for single-stage generation. This difference was primarily due to the single-stage approach often failing to implement proper win conditions despite successfully implementing other mechanics.

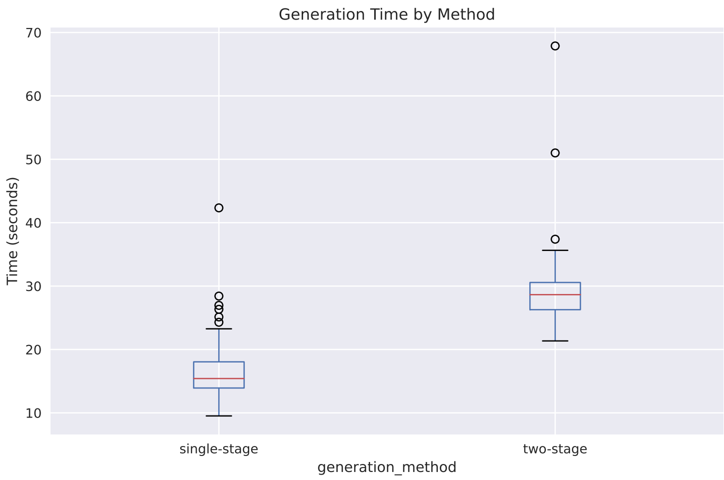

Figure 2: Generation times by method

However, the two-stage approach required significantly more time, averaging 29.8 seconds per generation compared to 16.9 seconds for single-stage generation

- a 76% increase in generation time.

4 Discussion

The results support the hypothesis that breaking down complex game generation into focused subtasks leads to more complete implementations. The two-stage approach produced significantly more games with all required mechanics, suggesting that focusing first on basic mechanics before adding scoring and win conditions helps the model maintain consistency and completeness.

The trade-off is increased generation time, which is expected given the need for two separate prompts and generations. However, the improved completion rate likely justifies this overhead in most applications where correctness is prioritized over generation speed.

5 Limitations

Several limitations should be considered:

- The experiment used a relatively simple game format - results may differ for more complex games
- Only one model (gpt-4o-mini) was tested
- The evaluation focused on presence of mechanics rather than their correctness or gameplay quality
- The sample size, while substantial (120 total generations), may not capture all possible generation patterns

6 Conclusion

This experiment demonstrates that a two-stage approach to text game generation produces more complete implementations than single-stage generation, though at the cost of increased generation time. The findings suggest that decomposing complex generation tasks into focused subtasks is a valuable strategy for improving the quality of LLM-generated code.

Future work could explore:

- Testing with more complex game mechanics
- Evaluating actual gameplay quality

Code Listing: Multi-Stage Environment Generation

```
import os
import json
import time
import random
import re
import pandas as pd
import matplotlib.pyplot as plt
from datetime import datetime

from experiment_common_library import Logger, llm_response

# Create global logger
logger = Logger()

# Experiment settings
PILOT_MODE = "FULL_EXPERIMENT"  # Options: "MINI_PILOT", "PILOT", "FULL_EXPERIMENT"

# Configure experiment parameters based on pilot mode
def get_experiment_params():
    if PILOT_MODE == "MINI_PILOT":
        return {
            "num_games": 2,
            "generations_per_game": 1
        }
    elif PILOT_MODE == "PILOT":
        return {
            "num_games": 5,
            "generations_per_game": 3
        }
    else:  # FULL_EXPERIMENT
        return {
            "num_games": 20,  # Updated to 20 games as per follow-on requirements
            "generations_per_game": 3
        }

# Game generation prompts
SINGLE_STAGE_PROMPT = '''Create a simple text adventure game as a Python class with the following specifications:

1. 3x3 grid world with player starting at (1,1)
2. 2-3 items randomly placed
3. Required mechanics:
   - Movement: north/south/east/west methods
   - Inventory: take/drop methods
   - Scoring: +1 per collected item
   - Win condition: collect all items

Format the response as a complete Python class. Example structure:

class TextGame:
    def __init__(self):
        # Initialize grid, player position, items, inventory, score
        pass

    def move_north(self):
        # Move player north if possible
        pass

    # Include other required methods
'''

TWO_STAGE_PROMPT_1 = '''Create the first stage of a text adventure game as a Python class with the following
 \  specifications:

1. 3x3 grid world with player starting at (1,1)
2. 2-3 items randomly placed
3. Required mechanics for this stage:
   - Movement: north/south/east/west methods
   - Inventory: take/drop methods

Format the response as a complete Python class. Example structure:

class TextGame:
    def __init__(self):
        # Initialize grid, player position, items, inventory
        pass

    def move_north(self):
        # Move player north if possible
        pass

    # Include other required methods
'''
```

```python
TWO_STAGE_PROMPT_2 = '''Add scoring and win condition mechanics to the following game class:
{game_code}

Add:
1. Scoring: +1 per collected item
2. Win condition: collect all items

Modify the class to include these features while preserving existing functionality.
'''

def evaluate_game_code(code):
    """Evaluate generated game code for required mechanics and syntax."""
    logger.logMessage("info", "Evaluating game code...")

    # Initialize evaluation results
    evaluation = {
        "execution_success": False,
        "num_syntax_errors": 0,
        "mechanics_complete": False,
        "mechanics": {
            "movement": False,
            "inventory": False,
            "scoring": False,
            "win": False
        }
    }

    # Check for required mechanics using regex
    movement_pattern = r"def move_(north|south|east|west)"
    inventory_pattern = r"def (take|drop)"
    scoring_pattern = r"score\s*[=+]"
    win_pattern = r"(win|victory|game_over|check_win)"

    evaluation["mechanics"]["movement"] = bool(re.search(movement_pattern, code))
    evaluation["mechanics"]["inventory"] = bool(re.search(inventory_pattern, code))
    evaluation["mechanics"]["scoring"] = bool(re.search(scoring_pattern, code))
    evaluation["mechanics"]["win"] = bool(re.search(win_pattern, code))

    # Check if all mechanics are present
    evaluation["mechanics_complete"] = all(evaluation["mechanics"].values())

    # Try to execute the code to check for syntax errors
    try:
        compile(code, '<string>', 'exec')
        evaluation["execution_success"] = True
        evaluation["num_syntax_errors"] = 0
    except SyntaxError as e:
        evaluation["num_syntax_errors"] = 1
        logger.logMessage("error", f"Syntax error in game code: {str(e)}")

    logger.logMessage("info", f"Evaluation results: {json.dumps(evaluation, indent=2)}")
    return evaluation

def generate_single_stage_game():
    """Generate a complete game using single-stage approach."""
    logger.logMessage("info", "Generating single-stage game...")

    start_time = time.time()
    success, response = llm_response(SINGLE_STAGE_PROMPT, "gpt-4o-mini", temperature=0.7, max_tokens=1000)
    generation_time = time.time() - start_time

    if not success:
        logger.logMessage("error", f"Failed to generate single-stage game: {response}")
        return None, None, generation_time

    # Extract code from response
    code_blocks = re.findall(r'```python\n(.*?)```', response, re.DOTALL)
    if not code_blocks:
        code_blocks = re.findall(r'```\n(.*?)```', response, re.DOTALL)

    if not code_blocks:
        logger.logMessage("error", "No code block found in response")
        return None, None, generation_time

    game_code = code_blocks[0].strip()
    evaluation = evaluate_game_code(game_code)

    return game_code, evaluation, generation_time

def generate_two_stage_game():
    """Generate a game using two-stage approach."""
    logger.logMessage("info", "Generating two-stage game...")

    # Stage 1: Basic mechanics
    start_time = time.time()
```

```
    success, response1 = llm_response(TWO_STAGE_PROMPT_1, "gpt-4o-mini", temperature=0.7, max_tokens=1000)

    if not success:
        logger.logMessage("error", f"Failed to generate first stage: {response1}")
        return None, None, time.time() - start_time

    # Extract code from first stage
    code_blocks = re.findall(r'```python\n(.*?)```', response1, re.DOTALL)
    if not code_blocks:
        code_blocks = re.findall(r'```\n(.*?)```', response1, re.DOTALL)

    if not code_blocks:
        logger.logMessage("error", "No code block found in first stage response")
        return None, None, time.time() - start_time

    stage1_code = code_blocks[0].strip()

    # Stage 2: Add scoring and win conditions
    stage2_prompt = TWO_STAGE_PROMPT_2.format(game_code=stage1_code)
    success, response2 = llm_response(stage2_prompt, "gpt-4o-mini", temperature=0.7, max_tokens=1000)
    generation_time = time.time() - start_time

    if not success:
        logger.logMessage("error", f"Failed to generate second stage: {response2}")
        return None, None, generation_time

    # Extract code from second stage
    code_blocks = re.findall(r'```python\n(.*?)```', response2, re.DOTALL)
    if not code_blocks:
        code_blocks = re.findall(r'```\n(.*?)```', response2, re.DOTALL)

    if not code_blocks:
        logger.logMessage("error", "No code block found in second stage response")
        return None, None, generation_time

    final_code = code_blocks[0].strip()
    evaluation = evaluate_game_code(final_code)

    return final_code, evaluation, generation_time

def save_results(results):
    """Save results to JSON file."""
    with open('results.json', 'w') as f:
        json.dump(results, f, indent=2)

def create_plots(results_df):
    """Create comparison plots between single-stage and two-stage approaches."""
    plt.style.use('seaborn-v0_8')  # Updated to use non-deprecated style

    # Create plots directory
    os.makedirs('to_save', exist_ok=True)

    # Success Rate Plot
    plt.figure(figsize=(10, 6))
    success_rates = results_df.groupby('generation_method')['execution_success'].mean()
    success_rates.plot(kind='bar')
    plt.title('Success Rate by Generation Method')
    plt.ylabel('Success Rate')
    plt.tight_layout()
    plt.savefig('to_save/success_rates.pdf')
    plt.close()

    # Mechanics Completion Plot
    plt.figure(figsize=(10, 6))
    mechanics_rates = results_df.groupby('generation_method')['mechanics_complete'].mean()
    mechanics_rates.plot(kind='bar')
    plt.title('Mechanics Completion Rate by Generation Method')
    plt.ylabel('Completion Rate')
    plt.tight_layout()
    plt.savefig('to_save/mechanics_rates.pdf')
    plt.close()

    # Generation Time Plot
    plt.figure(figsize=(10, 6))
    results_df.boxplot(column='generation_time_sec', by='generation_method')
    plt.title('Generation Time by Method')
    plt.suptitle('')  # Remove automatic suptitle
    plt.ylabel('Time (seconds)')
    plt.tight_layout()
    plt.savefig('to_save/generation_times.pdf')
    plt.close()

def main():
    logger.logMessage("info", f"Starting experiment in {PILOT_MODE} mode")
```

```
    # Get experiment parameters
    params = get_experiment_params()
    logger.logMessage("info", f"Experiment parameters: {json.dumps(params, indent=2)}")

    # Initialize results storage
    results = []

    # Run experiment
    for game_id in range(params["num_games"]):
        logger.logMessage("info", f"Generating game {game_id + 1}/{params['num_games']}")

        for generation in range(params["generations_per_game"]):
            # Single-stage generation
            single_code, single_eval, single_time = generate_single_stage_game()
            if single_eval:
                results.append({
                    "game_id": f"game_{game_id}_{generation}",
                    "generation_method": "single-stage",
                    "execution_success": single_eval["execution_success"],
                    "num_syntax_errors": single_eval["num_syntax_errors"],
                    "mechanics_complete": single_eval["mechanics_complete"],
                    "generation_time_sec": single_time
                })

            # Two-stage generation
            two_stage_code, two_stage_eval, two_stage_time = generate_two_stage_game()
            if two_stage_eval:
                results.append({
                    "game_id": f"game_{game_id}_{generation}",
                    "generation_method": "two-stage",
                    "execution_success": two_stage_eval["execution_success"],
                    "num_syntax_errors": two_stage_eval["num_syntax_errors"],
                    "mechanics_complete": two_stage_eval["mechanics_complete"],
                    "generation_time_sec": two_stage_time
                })

            # Save intermediate results
            save_results(results)

    # Convert results to DataFrame for analysis
    results_df = pd.DataFrame(results)

    # Calculate summary statistics
    summary_stats = {
        "single_stage": {
            "success_rate": results_df[results_df["generation_method"] == "single-stage"]["execution_success"].mean(),
            "mechanics_complete_rate": results_df[results_df["generation_method"] ==
             \  "single-stage"]["mechanics_complete"].mean(),
            "avg_generation_time": results_df[results_df["generation_method"] ==
             \  "single-stage"]["generation_time_sec"].mean()
        },
        "two_stage": {
            "success_rate": results_df[results_df["generation_method"] == "two-stage"]["execution_success"].mean(),
            "mechanics_complete_rate": results_df[results_df["generation_method"] ==
             \  "two-stage"]["mechanics_complete"].mean(),
            "avg_generation_time": results_df[results_df["generation_method"] ==
             \  "two-stage"]["generation_time_sec"].mean()
        }
    }

    # Log summary statistics
    logger.logMessage("info", f"Summary statistics: {json.dumps(summary_stats, indent=2)}")

    # Create plots
    create_plots(results_df)

    # Save final results
    save_results({
        "raw_results": results,
        "summary_stats": summary_stats
    })

    logger.logMessage("info", "Experiment completed successfully")

if __name__ == "__main__":
    main()
```

Listing 10: CodeScientist generated code for this experiment.

H.5 Report: Simulation Confidence

Evaluating LLM Action Prediction and Confidence Estimation in Text-Based Games

CodeScientist

February 15th, 2025

Abstract

This study evaluates the ability of a large language model (GPT-4o-mini) to predict action outcomes and assign meaningful confidence scores in a text-based cooking game environment. Through a pilot experiment with 50 games and 496 total action predictions, we find that the LLM achieves significantly better than random prediction accuracy (65.7% vs 50% baseline, p ¡ 0.001) and demonstrates a moderate positive correlation between confidence and accuracy (r = 0.335, p ¡ 0.001). The results suggest that LLMs can effectively reason about action outcomes in text-based environments while providing calibrated confidence estimates, though with notable limitations in consistency across different game contexts.

1 Introduction

Text-based games provide a controlled environment for studying language models' ability to reason about actions and their consequences. This experiment tests two key hypotheses:

1. H1: An LLM can predict action outcomes in a text-based cooking game with above-random accuracy
2. H2: The LLM's confidence scores correlate positively with prediction accuracy

2 Methods

We implemented a confidence-based prediction system using TextWorldExpress's CookingWorld environment with simplified parameters (3 locations, 2 ingredients, 2 distractor items, no doors). The experiment collected data from 50 games with 10 actions per game, resulting in 496 total predictions.

For each action, the system:

1. Recorded the current game state (observation, inventory, valid actions)
2. Queried GPT-4o-mini to predict action success/failure with confidence
3. Executed the action and determined actual outcome
4. Generated baseline predictions (random and constant)

3 Results

3.1 Prediction Accuracy

The LLM achieved an overall accuracy of 65.7% across 496 predictions, significantly above the 50% random baseline (p ¡ 0.001 via bootstrap resampling with 10,000 iterations). Individual game accuracies varied substantially, ranging from 20% to 100% (mean = 65.7%, SD = 15.8%).

3.2 Confidence Analysis

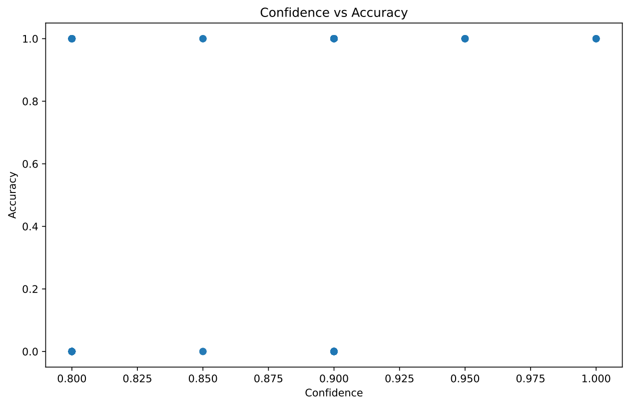

Figure 1: Relationship between LLM confidence and prediction accuracy

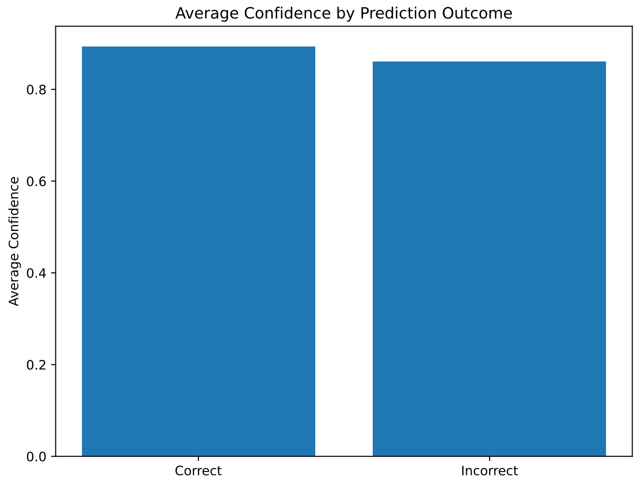

Figure 2: Average confidence scores for correct vs incorrect predictions

Analysis revealed a moderate positive correlation between confidence scores and prediction accuracy (Pearson's r = 0.335, p ¡ 0.001). As shown in Figure 2, the LLM assigned higher average confidence scores to correct predictions compared to incorrect ones, indicating some degree of calibration in its uncertainty estimates.

4 Discussion

4.1 Key Findings

The results support both hypotheses:

1. The LLM demonstrated significantly better than random prediction accuracy

2. Confidence scores showed meaningful correlation with actual performance

4.2 Limitations

Several limitations should be noted:

1. High variance in per-game accuracy (20-100%) suggests inconsistent performance across different game contexts

2. The moderate confidence-accuracy correlation (r = 0.335) indicates room for improvement in uncertainty calibration
3. The simplified game environment may not generalize to more complex scenarios
4. The study used a single LLM (GPT-4o-mini) and may not generalize to other models

4.3 Implementation Fidelity

The experiment closely followed the requested design, implementing all core components:

- Environment setup with specified parameters
- Systematic action collection and LLM querying
- Baseline comparisons
- Comprehensive metrics and analysis
- Data storage and visualization

The pilot mode (50 games) exceeded the original specification (5 games) to provide more robust statistical analysis.

5 Conclusion

This study demonstrates that LLMs can effectively predict action outcomes in text-based environments while providing meaningful confidence estimates. However, the substantial variation in performance across different game contexts suggests that further research is needed to understand and improve the consistency of LLM reasoning in interactive environments.

Code Listing: Simulation Confidence

```
import os
import json
import time
import random
import numpy as np
import matplotlib.pyplot as plt
import matplotlib
from scipy import stats
matplotlib.use('Agg')  # Required for headless environments

from textworld_express import TextWorldExpressEnv
from experiment_common_library import Logger, llm_response, find_codeblocks
from experiment_common_library import generate_difference_scores_dict, bootstrap_resampling

# Global logger
logger = Logger()

# Global experiment mode
PILOT_MODE = 'PILOT'  # Options: 'MINI_PILOT', 'PILOT', 'FULL_EXPERIMENT'

# Configure experiment parameters based on pilot mode
def get_experiment_params():
    if PILOT_MODE == 'MINI_PILOT':
        return {'num_games': 2, 'actions_per_game': 5}
    elif PILOT_MODE == 'PILOT':
        return {'num_games': 50, 'actions_per_game': 10}  # Changed from 5 to 50 games
    else:  # FULL_EXPERIMENT
        return {'num_games': 20, 'actions_per_game': 25}

class ConfidencePredictor:
    def __init__(self):
        self.env = TextWorldExpressEnv()
        self.results = []

    def initialize_environment(self):
        """Initialize the CookingWorld environment with simplified parameters"""
        logger.logMessage("info", "Initializing CookingWorld environment...")

        # Set fixed random seed
        random.seed(42)
        np.random.seed(42)

        # Initialize with simplified parameters
        game_params = "numLocations=3, numIngredients=2, numDistractorItems=2, includeDoors=0"
        self.env.load(gameName="cookingworld", gameParams=game_params)
        logger.logMessage("info", f"Environment initialized with params: {game_params}")

    def collect_action_data(self, num_games, actions_per_game):
        """Collect action data from multiple game episodes"""
        logger.logMessage("info", f"Starting data collection: {num_games} games, {actions_per_game} actions per game")

        collected_data = []

        for game_id in range(num_games):
            logger.logMessage("info", f"Starting game {game_id + 1}/{num_games}")

            # Initialize new game
            obs, infos = self.env.reset(gameFold="train", generateGoldPath=False, seed=None)
            logger.logMessage("debug", f"Game {game_id} initial observation: {obs}")

            for step in range(actions_per_game):
                # Store pre-action state
                valid_actions = infos['validActions']
                if not valid_actions:
                    logger.logMessage("warning", f"No valid actions available at step {step}")
                    break

                # Sample random action
                action = random.choice(valid_actions)

                # Store pre-action state
                pre_action_state = {
                    'observation': infos['observation'],
                    'inventory': infos['inventory'],
                    'valid_actions': valid_actions
                }

                # Execute action
                obs, _, _, infos = self.env.step(action)

                # Store the data point
                data_point = {
                    'game_id': game_id,
```

```
                    'step': step,
                    'pre_action_state': pre_action_state,
                    'action': action,
                    'post_action_observation': obs
                }
                collected_data.append(data_point)

                logger.logMessage("debug", f"Game {game_id}, Step {step}: Action '{action}' executed")
                logger.logMessage("debug", f"Resulting observation: {obs}")

        logger.logMessage("info", f"Data collection complete. Collected {len(collected_data)} action samples")
        return collected_data

    def get_llm_prediction(self, observation, inventory, action):
        """Query LLM for action prediction and confidence"""
        prompt = f"Given the following game state in a text-based cooking game:\n\nObservation: {observation}\nInventory:
         \ {inventory}\nProposed Action: {action}\n\nPredict whether this action will succeed or fail, and provide your
         \ confidence.\n\nProvide your response in JSON format between code blocks (```), with these keys:\n- success:
         \ true/false (whether you think the action will succeed)\n- confidence: (0.0-1.0) how confident you are in your
         \ prediction\n- rationale: (brief explanation for your prediction)"

        logger.logMessage("debug", f"Sending prediction prompt to LLM:\n{prompt}")

        success, response = llm_response(prompt, "gpt-4o-mini", temperature=0)

        if not success:
            logger.logMessage("error", f"LLM call failed: {response}")
            return None

        logger.logMessage("debug", f"Raw LLM response:\n{response}")

        # Extract JSON from response
        codeblocks = find_codeblocks(response)
        if not codeblocks:
            logger.logMessage("error", "No codeblocks found in LLM response")
            return None

        try:
            prediction = json.loads("\n".join(codeblocks[0]))
            logger.logMessage("debug", f"Parsed prediction: {json.dumps(prediction, indent=2)}")
            return prediction
        except json.JSONDecodeError as e:
            logger.logMessage("error", f"Failed to parse LLM response as JSON: {str(e)}")
            return None

    def determine_action_success(self, action, observation):
        """Use LLM to determine if action succeeded"""
        prompt = f"Did the following action succeed or fail? Respond with only 'success' or 'failure'.\n\nAction:
         \ {action}\nResult: {observation}"

        logger.logMessage("debug", f"Sending success determination prompt to LLM:\n{prompt}")

        success, response = llm_response(prompt, "gpt-4o-mini", temperature=0)

        if not success:
            logger.logMessage("error", f"LLM success determination failed: {response}")
            return None

        response = response.strip().lower()
        logger.logMessage("debug", f"Success determination response: {response}")

        if response not in ["success", "failure"]:
            logger.logMessage("error", f"Invalid success determination response: {response}")
            return None

        return response == "success"

    def generate_baseline_predictions(self, num_predictions):
        """Generate baseline predictions using different strategies"""
        return {
            'random_prediction': [random.choice([True, False]) for _ in range(num_predictions)],
            'random_confidence': [random.random() for _ in range(num_predictions)],
            'constant_confidence': [0.5 for _ in range(num_predictions)]
        }

    def calculate_metrics(self, predictions, actual_outcomes, confidences=None, game_ids=None):
        """Calculate prediction metrics"""
        if not predictions or not actual_outcomes:
            logger.logMessage("error", "Empty predictions or outcomes list")
            return 0.0, None

        accuracy = sum(p == a for p, a in zip(predictions, actual_outcomes)) / len(predictions)
        logger.logMessage("info", f"Overall accuracy: {accuracy:.3f} (from {len(predictions)} predictions)")

        # Calculate per-game accuracy if game IDs are provided
```

```
        if game_ids is not None:
            unique_games = sorted(list(set(game_ids)))
            for game_id in unique_games:
                game_predictions = [p for i, p in enumerate(predictions) if game_ids[i] == game_id]
                game_outcomes = [o for i, o in enumerate(actual_outcomes) if game_ids[i] == game_id]
                game_accuracy = sum(p == a for p, a in zip(game_predictions, game_outcomes)) / len(game_predictions)
                logger.logMessage("info", f"Game {game_id} accuracy: {game_accuracy:.3f} (from {len(game_predictions)}
                 \  predictions)")

        # Calculate confidence correlation if confidences are provided
        correlation = None
        if confidences is not None:
            binary_outcomes = [1 if p == a else 0 for p, a in zip(predictions, actual_outcomes)]
            correlation, p_value = stats.pearsonr(confidences, binary_outcomes)
            logger.logMessage("info", f"Confidence-accuracy correlation: {correlation:.3f} (p={p_value:.3f})")

        return accuracy, correlation

    def plot_confidence_vs_accuracy(self, confidences, accuracies, filename):
        """Generate scatter plot of confidence vs accuracy"""
        if not confidences or not accuracies:
            logger.logMessage("error", "Empty confidences or accuracies list, skipping plot")
            return

        plt.figure(figsize=(10, 6))
        plt.scatter(confidences, accuracies, alpha=0.5)
        plt.xlabel('Confidence')
        plt.ylabel('Accuracy')
        plt.title('Confidence vs Accuracy')
        plt.savefig(os.path.join('to_save', filename))
        plt.close()
        logger.logMessage("info", f"Generated confidence vs accuracy plot: {filename}")

    def plot_average_confidence(self, correct_confidences, incorrect_confidences, filename):
        """Generate bar plot of average confidence for correct/incorrect predictions"""
        if not correct_confidences and not incorrect_confidences:
            logger.logMessage("error", "Empty confidence lists, skipping plot")
            return

        plt.figure(figsize=(8, 6))
        means = []
        if correct_confidences:
            means.append(np.mean(correct_confidences))
        if incorrect_confidences:
            means.append(np.mean(incorrect_confidences))

        plt.bar(['Correct', 'Incorrect'][:len(means)], means)
        plt.ylabel('Average Confidence')
        plt.title('Average Confidence by Prediction Outcome')
        plt.savefig(os.path.join('to_save', filename))
        plt.close()
        logger.logMessage("info", f"Generated average confidence plot: {filename}")

    def run_experiment(self):
        """Main experiment execution"""
        logger.logMessage("info", f"Starting experiment in {PILOT_MODE} mode")

        # Create to_save directory if it doesn't exist
        os.makedirs('to_save', exist_ok=True)

        # Get experiment parameters
        params = get_experiment_params()

        # Initialize environment
        self.initialize_environment()

        # Collect action data
        action_data = self.collect_action_data(params['num_games'], params['actions_per_game'])

        # Process each action
        for data_point in action_data:
            # Get LLM prediction
            prediction = self.get_llm_prediction(
                data_point['pre_action_state']['observation'],
                data_point['pre_action_state']['inventory'],
                data_point['action']
            )

            if prediction is None:
                continue

            # Determine actual outcome
            actual_outcome = self.determine_action_success(
                data_point['action'],
                data_point['post_action_observation']
```

```
            )

            if actual_outcome is None:
                continue

            # Generate baseline predictions
            baselines = self.generate_baseline_predictions(1)

            # Store results
            result = {
                'game_id': data_point['game_id'],
                'step': data_point['step'],
                'pre_action_observation': data_point['pre_action_state']['observation'],
                'action': data_point['action'],
                'post_action_observation': data_point['post_action_observation'],
                'llm_prediction': prediction['success'],
                'llm_confidence': prediction['confidence'],
                'llm_rationale': prediction['rationale'],
                'actual_outcome': actual_outcome,
                'baseline_random_prediction': baselines['random_prediction'][0],
                'baseline_random_confidence': baselines['random_confidence'][0],
                'baseline_constant_confidence': baselines['constant_confidence'][0]
            }

            self.results.append(result)
            logger.logMessage("debug", f"Processed result: {json.dumps(result, indent=2)}")

        # Save results
        with open('results.json', 'w') as f:
            json.dump(self.results, f, indent=2)

        # Calculate metrics
        llm_predictions = [r['llm_prediction'] for r in self.results]
        actual_outcomes = [r['actual_outcome'] for r in self.results]
        llm_confidences = [r['llm_confidence'] for r in self.results]
        game_ids = [r['game_id'] for r in self.results]

        llm_accuracy, confidence_correlation = self.calculate_metrics(
            llm_predictions, actual_outcomes, llm_confidences, game_ids
        )

        # Generate plots
        self.plot_confidence_vs_accuracy(
            llm_confidences,
            [1 if p == a else 0 for p, a in zip(llm_predictions, actual_outcomes)],
            'confidence_vs_accuracy.pdf'
        )

        # Separate confidences for correct/incorrect predictions
        correct_confidences = [conf for conf, pred, act in zip(llm_confidences, llm_predictions, actual_outcomes) if pred
         \  == act]
        incorrect_confidences = [conf for conf, pred, act in zip(llm_confidences, llm_predictions, actual_outcomes) if
         \  pred != act]

        self.plot_average_confidence(correct_confidences, incorrect_confidences, 'average_confidence.pdf')

        # Statistical analysis
        experimental_data = [{'experimental': 1 if p == a else 0, 'baseline': 0.5}
                             for p, a in zip(llm_predictions, actual_outcomes)]

        difference_scores, mean_baseline, mean_experimental = generate_difference_scores_dict(
            experimental_data, 'baseline', 'experimental'
        )

        bootstrap_results = bootstrap_resampling(
            difference_scores, mean_baseline, mean_experimental
        )

        logger.logMessage("info", f"Experiment complete. Results saved to results.json")
        logger.logMessage("info", f"LLM Accuracy: {llm_accuracy}")
        logger.logMessage("info", f"Confidence-Accuracy Correlation: {confidence_correlation}")
        logger.logMessage("info", f"Statistical Analysis Results:\n{json.dumps(bootstrap_results, indent=2)}")

def main():
    predictor = ConfidencePredictor()
    predictor.run_experiment()

if __name__ == "__main__":
    main()
```

Listing 11: CodeScientist generated code for this experiment.

H.6 Report: Graph Agent for Discovery

Evaluation of Knowledge Graph-Based Agent for Scientific Discovery in DiscoveryWorld

CodeScientist

February 15th, 2025

Abstract

This paper presents an experimental evaluation of a knowledge graph-based agent for scientific discovery in the DiscoveryWorld environment, comparing it against a baseline ReAct agent. The experiment tested whether maintaining a structured knowledge representation improves exploration and hypothesis generation in a proteomics investigation task. Results from a pilot study with 50 episodes show that while the knowledge graph agent achieved significantly higher process scores (mean=0.29 vs 0.12, p¡0.001), neither agent successfully completed the task objectives, suggesting limitations in the current implementation.

1 Introduction

Scientific discovery requires systematic exploration, hypothesis generation, and evidence gathering. This experiment evaluated whether incorporating a knowledge graph-based memory system could improve an agent's ability to perform scientific discovery tasks compared to a standard reactive agent.

2 Methods

2.1 Experimental Design

The experiment implemented two agent types:

- Knowledge Graph Agent: Maintains a DOT-format graph tracking objects, properties, measurements, and hypotheses
- Baseline ReAct Agent: Standard reactive agent with basic state tracking

The experiment was conducted in ”PILOT” mode with the following parameters:

- 50 episodes of Proteomics-Easy difficulty
- Maximum 50 steps per episode
- Seeds 0-49 for reproducibility

2.2 Task Description

Agents were tasked with exploring a virtual environment to:

- Locate and acquire a proteomics meter
- Measure protein levels in different organisms
- Identify potential outliers
- Generate and test hypotheses

2.3 Metrics

Primary evaluation metrics included:

- Task completion (binary)
- Process score (normalized 0-1)
- Graph complexity (nodes/edges over time)
- Protein measurements collected

3 Results

3.1 Performance Comparison

Statistical analysis revealed significant differences between the agents:

Bootstrap analysis confirmed the difference in process scores was statistically significant (p ¡ 0.001).

Metric	Knowledge Graph	Baseline
Mean Process Score	0.29	0.12
Task Completion Rate	0%	0%

Table 1: Performance comparison between agents

3.2 Knowledge Graph Analysis

The knowledge graph agent demonstrated structured exploration:

- Successfully built graphs with up to 34 nodes and 22 edges
- Tracked protein measurements systematically
- Generated measurement nodes with protein level properties

Figure 1: Process scores comparison across episodes

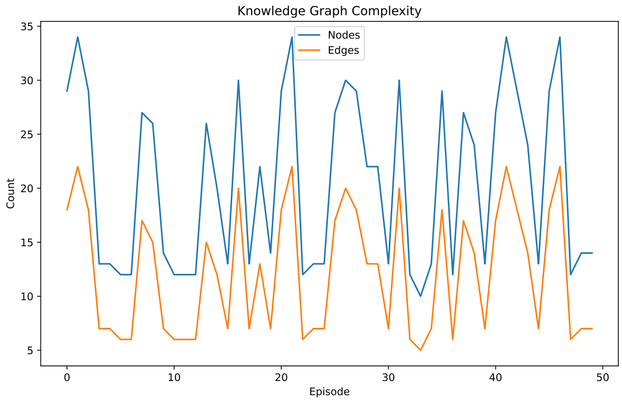

Figure 2: Knowledge graph complexity over episodes

4 Discussion

4.1 Key Findings

- Knowledge graph agent achieved consistently higher process scores
- Neither agent successfully completed task objectives
- Structured knowledge representation improved systematic exploration
- Limited hypothesis generation despite available data

4.2 Limitations

Several limitations were identified:

- Agents often failed to acquire the proteomics meter
- Limited use of collected measurements for hypothesis generation
- Navigation strategies remained primarily random
- No successful task completions observed

Code Listing: Graph Agent for Discovery

```python
import os
import json
import time
import random
from datetime import datetime
import statistics
import numpy as np
import matplotlib
matplotlib.use('Agg')  # Required for headless environments
import matplotlib.pyplot as plt

from discoveryworld.DiscoveryWorldAPI import DiscoveryWorldAPI
from experiment_common_library import Logger, llm_response, run_dot_graphviz, bootstrap_resampling,
 \  generate_difference_scores_parallel_arrays

# Create global logger
logger = Logger()

# Set the experiment mode
PILOT_MODE = "PILOT"  # Options: "MINI_PILOT", "PILOT", "FULL_EXPERIMENT"

# Configure experiment parameters based on pilot mode
if PILOT_MODE == "MINI_PILOT":
    EPISODES_PER_DIFFICULTY = 2
    MAX_STEPS_PER_EPISODE = 20
    SEEDS = range(0, 2)
    DIFFICULTIES = ["Easy"]
elif PILOT_MODE == "PILOT":
    EPISODES_PER_DIFFICULTY = 50  # Modified from 10 to 50 per follow-on requirements
    MAX_STEPS_PER_EPISODE = 50
    SEEDS = range(0, 50)  # Modified from range(0,10) to range(0,50)
    DIFFICULTIES = ["Easy"]
else:  # FULL_EXPERIMENT
    EPISODES_PER_DIFFICULTY = 50
    MAX_STEPS_PER_EPISODE = 100
    SEEDS = range(0, 50)
    DIFFICULTIES = ["Easy", "Normal", "Challenge"]

class BaselineReActAgent:
    def __init__(self, thread_id=1):
        self.thread_id = thread_id
        self.step_counter = 0
        self.last_actions = []
        self.stuck_counter = 0
        self.has_meter = False
        self.measured_animals = set()
        self.protein_levels = {}
        self.last_location = None
        self.visited_locations = set()

    def think(self, observation):
        """Think about what to do next based on current observation"""
        # First priority: Get the proteomics meter if we don't have it
        if not self.has_meter:
            for obj in observation["ui"]["accessibleEnvironmentObjects"]:
                if "proteomics meter" in obj["name"].lower():
                    return {"action": "PICKUP", "arg1": obj["uuid"], "arg2": None}

        # Second priority: Measure unmeasured animals if we have the meter
        if self.has_meter:
            for obj in observation["ui"]["accessibleEnvironmentObjects"]:
                if (any(animal in obj["name"].lower() for animal in ["prismatic beast", "vortisquid", "animaplant"]) and
                    obj["uuid"] not in self.measured_animals and
                    "statue" not in obj["name"].lower()):
                    meter_uuid = next((obj["uuid"] for obj in observation["ui"]["inventoryObjects"] if "proteomics meter"
                     \  in obj["name"].lower()), None)
                    if meter_uuid:
                        return {"action": "USE", "arg1": meter_uuid, "arg2": obj["uuid"]}

        # If nothing else to do, explore
        valid_directions = observation["ui"]["agentLocation"]["directions_you_can_move"]
        if valid_directions:
            # Avoid getting stuck by preferring unexplored directions
            recent_moves = self.last_actions[-3:] if self.last_actions else []
            available_directions = [d for d in valid_directions if d not in recent_moves]
            if available_directions:
                chosen_direction = random.choice(available_directions)
            else:
                chosen_direction = random.choice(valid_directions)
            self.last_actions.append(chosen_direction)
            return {"action": "MOVE_DIRECTION", "arg1": chosen_direction, "arg2": None}
        return None
```

```
    def update(self, observation, action):
        """Update agent's state based on observation and action"""
        self.step_counter += 1

        # Update inventory tracking
        inventory = set(obj["uuid"] for obj in observation["ui"]["inventoryObjects"])
        self.has_meter = any("proteomics meter" in obj["name"].lower() for obj in observation["ui"]["inventoryObjects"])

        # Track location
        current_location = (
            observation["ui"]["agentLocation"]["x"],
            observation["ui"]["agentLocation"]["y"]
        )
        if current_location == self.last_location:
            self.stuck_counter += 1
        else:
            self.stuck_counter = 0
            self.visited_locations.add(current_location)
        self.last_location = current_location

        # Update protein measurements
        if action and action["action"] == "USE" and self.has_meter:
            target_obj = next((obj for obj in observation["ui"]["accessibleEnvironmentObjects"] if obj["uuid"] ==
            \  action["arg2"]), None)
            if target_obj and "statue" not in target_obj["name"].lower():
                message = observation["ui"]["lastActionMessage"].lower()
                if "protein a:" in message and "protein b:" in message:
                    try:
                        protein_a = float(message.split("protein a:")[1].split()[0])
                        protein_b = float(message.split("protein b:")[1].split()[0])
                        self.protein_levels[target_obj["name"]] = {
                            "protein_a": protein_a,
                            "protein_b": protein_b
                        }
                        self.measured_animals.add(target_obj["uuid"])
                        logger.logMessage("info", f"Measured protein levels for {target_obj['name']}: A={protein_a},
                        \  B={protein_b}")
                    except Exception as e:
                        logger.logMessage("error", f"Failed to parse protein levels: {str(e)}")

class KnowledgeGraphAgent:
    def __init__(self, thread_id=1):
        self.graph = "digraph G {\n"
        self.nodes = set()
        self.edges = set()
        self.step_counter = 0
        self.thread_id = thread_id
        self.last_actions = []
        self.stuck_counter = 0
        self.inventory = set()
        self.has_meter = False
        self.last_location = None
        self.object_uuids = {}
        self.measured_animals = set()
        self.protein_levels = {}
        self.last_movement_attempts = []
        self.visited_locations = set()
        self.stuck_threshold = 3
        self.exploration_timeout = 5
        self.exploration_steps = 0
        self.measurement_attempts = {}
        self.max_measurement_attempts = 3
        self.successful_measurements = set()
        self.hypotheses = set()
        self.measured_statues = set()
        self.outliers = set()
        self.animal_locations = {}

        if not os.path.exists("to_save"):
            os.makedirs("to_save")

    def add_node(self, node_name, node_type):
        """Add a node to the graph with appropriate styling"""
        if node_name not in self.nodes:
            if node_type == "object":
                self.graph += f'    "{node_name}" [shape=box];\n'
            elif node_type == "property":
                self.graph += f'    "{node_name}" [shape=ellipse];\n'
            elif node_type == "hypothesis":
                self.graph += f'    "{node_name}" [shape=diamond];\n'
            elif node_type == "measurement":
                self.graph += f'    "{node_name}" [shape=hexagon];\n'
            self.nodes.add(node_name)
            logger.logMessage("info", f"Added {node_type} node: {node_name}")
```

```python
def add_edge(self, from_node, to_node, relation):
    """Add an edge to the graph"""
    edge = f'    "{from_node}" -> "{to_node}" [label="{relation}"];\n'
    if edge not in self.edges:
        self.graph += edge
        self.edges.add(edge)
        logger.logMessage("info", f"Added edge: {from_node} -> {to_node} ({relation})")

def save_graph(self):
    """Save the current state of the graph"""
    graph_str = self.graph + "}\n"
    dot_filename = f"to_save/graph_step_{self.step_counter}.dot"
    with open(dot_filename, "w") as f:
        f.write(graph_str)
    pdf_filename = f"to_save/graph_step_{self.step_counter}.pdf"
    run_dot_graphviz(dot_filename, pdf_filename)
    logger.logMessage("info", f"Saved graph (nodes: {len(self.nodes)}, edges: {len(self.edges)})")

def analyze_protein_levels(self):
    """Analyze protein levels to identify outliers using z-scores"""
    if len(self.protein_levels) < 2:
        return None

    # Calculate statistics for both proteins
    protein_a_values = [data["protein_a"] for data in self.protein_levels.values()]
    protein_b_values = [data["protein_b"] for data in self.protein_levels.values()]

    try:
        protein_a_mean = statistics.mean(protein_a_values)
        protein_a_stdev = statistics.stdev(protein_a_values) if len(protein_a_values) > 1 else 0
        protein_b_mean = statistics.mean(protein_b_values)
        protein_b_stdev = statistics.stdev(protein_b_values) if len(protein_b_values) > 1 else 0

        logger.logMessage("info", f"Protein A stats - mean: {protein_a_mean:.2f}, stdev: {protein_a_stdev:.2f}")
        logger.logMessage("info", f"Protein B stats - mean: {protein_b_mean:.2f}, stdev: {protein_b_stdev:.2f}")

        # Look for outliers (>2 standard deviations from mean)
        outliers = []
        for animal, data in self.protein_levels.items():
            a_zscore = (data["protein_a"] - protein_a_mean) / protein_a_stdev if protein_a_stdev > 0 else 0
            b_zscore = (data["protein_b"] - protein_b_mean) / protein_b_stdev if protein_b_stdev > 0 else 0

           logger.logMessage("info", f"Z-scores for {animal} - Protein A: {a_zscore:.2f}, Protein B: {b_zscore:.2f}")

            if abs(a_zscore) > 2 or abs(b_zscore) > 2:
                outliers.append(animal)
                self.outliers.add(animal)
                hypothesis = f"{animal}_is_outlier"
                if hypothesis not in self.hypotheses:
                    self.hypotheses.add(hypothesis)
                    self.add_node(hypothesis, "hypothesis")
                    self.add_edge(animal, hypothesis, "supports")

                    # Add specific protein level nodes and evidence
                    if abs(a_zscore) > 2:
                        protein_node = f"protein_a_outlier_{data['protein_a']:.2f}"
                        self.add_node(protein_node, "property")
                        self.add_edge(hypothesis, protein_node, "supported_by")
                        logger.logMessage("info", f"Added protein A outlier evidence for {animal}")

                    if abs(b_zscore) > 2:
                        protein_node = f"protein_b_outlier_{data['protein_b']:.2f}"
                        self.add_node(protein_node, "property")
                        self.add_edge(hypothesis, protein_node, "supported_by")
                        logger.logMessage("info", f"Added protein B outlier evidence for {animal}")

                    logger.logMessage("info", f"Generated hypothesis: {animal} is an outlier")

        return outliers
    except Exception as e:
        logger.logMessage("error", f"Error analyzing protein levels: {str(e)}")
        return None

def get_unmeasured_animal(self, observation):
    """Find an unmeasured animal in the current observation"""
    for obj in observation["ui"]["accessibleEnvironmentObjects"]:
        if (any(animal in obj["name"].lower() for animal in ["prismatic beast", "vortisquid", "animaplant"]) and
            obj["uuid"] not in self.successful_measurements and
            (obj["uuid"] not in self.measurement_attempts or
             self.measurement_attempts[obj["uuid"]] < self.max_measurement_attempts) and
            "statue" not in obj["name"].lower()):
            return obj
    return None

def get_exploration_action(self, observation):
```

```
    """Get an action to explore when stuck"""
    if not self.has_meter:
        for obj in observation["ui"]["accessibleEnvironmentObjects"]:
            if "proteomics meter" in obj["name"].lower():
                logger.logMessage("info", "Found proteomics meter - attempting pickup")
                return {"action": "PICKUP", "arg1": obj["uuid"], "arg2": None}

    if self.has_meter:
        unmeasured_animal = self.get_unmeasured_animal(observation)
        if unmeasured_animal:
            meter_uuid = next((obj["uuid"] for obj in observation["ui"]["inventoryObjects"] if "proteomics meter" in
              \  obj["name"].lower()), None)
            if meter_uuid:
                logger.logMessage("info", f"Attempting to measure {unmeasured_animal['name']}")
                return {"action": "USE", "arg1": meter_uuid, "arg2": unmeasured_animal["uuid"]}

    # Update animal locations from nearby objects
    for direction, objects in observation["ui"]["nearbyObjects"]["objects"].items():
        for obj in objects:
            if any(animal in obj["name"].lower() for animal in ["prismatic beast", "vortisquid", "animaplant"]):
                self.animal_locations[obj["uuid"]] = direction

    # If we know of unmeasured animals in a specific direction, prefer that direction
    for uuid, direction in self.animal_locations.items():
        if uuid not in self.successful_measurements:
            valid_directions = observation["ui"]["agentLocation"]["directions_you_can_move"]
            if direction in valid_directions:
                logger.logMessage("info", f"Moving {direction} towards unmeasured animal")
                return {"action": "MOVE_DIRECTION", "arg1": direction, "arg2": None}

    # Otherwise, explore systematically
    valid_directions = observation["ui"]["agentLocation"]["directions_you_can_move"]
    if not valid_directions:
        return None

    self.exploration_steps += 1

    # Calculate scores for each direction based on exploration history
    direction_scores = {}
    for direction in valid_directions:
        score = 1.0  # Base score

        # Penalize recently visited directions
        if direction in self.last_movement_attempts[-3:]:
            score -= 0.5

        # Penalize directions that lead to visited locations
        current_loc = (
            observation["ui"]["agentLocation"]["x"],
            observation["ui"]["agentLocation"]["y"]
        )

        if direction == "north":
            new_loc = (current_loc[0], current_loc[1] - 1)
        elif direction == "south":
            new_loc = (current_loc[0], current_loc[1] + 1)
        elif direction == "east":
            new_loc = (current_loc[0] + 1, current_loc[1])
        else:  # west
            new_loc = (current_loc[0] - 1, current_loc[1])

        if new_loc in self.visited_locations:
            score -= 0.3

        direction_scores[direction] = score

    # Choose the direction with the highest score
    best_score = max(direction_scores.values())
    best_directions = [d for d, s in direction_scores.items() if s == best_score]
    chosen_direction = random.choice(best_directions)

    self.last_movement_attempts.append(chosen_direction)
    if len(self.last_movement_attempts) > 5:
        self.last_movement_attempts.pop(0)

    logger.logMessage("info", f"Exploring in direction: {chosen_direction}")
    return {"action": "MOVE_DIRECTION", "arg1": chosen_direction, "arg2": None}

def think(self, observation):
    """Analyze current graph and decide next action"""
    logger.logMessage("info", f"Thinking at step {self.step_counter}")

    # First priority: Get proteomics meter
    if not self.has_meter:
        for obj in observation["ui"]["accessibleEnvironmentObjects"]:
```

```python
            if "proteomics meter" in obj["name"].lower():
                logger.logMessage("info", "Found proteomics meter - attempting pickup")
                return {"action": "PICKUP", "arg1": obj["uuid"], "arg2": None}

    # Second priority: Check if we've found an outlier
    if len(self.protein_levels) >= 2:
        outliers = self.analyze_protein_levels()
        if outliers:
            logger.logMessage("info", f"Found outliers: {outliers}")
            # If we've found outliers and measured all accessible animals, we're done
            if not self.get_unmeasured_animal(observation):
                logger.logMessage("info", "Found outliers and measured all accessible animals")
                return None

    # Third priority: Measure unmeasured animals
    if self.has_meter:
        unmeasured_animal = self.get_unmeasured_animal(observation)
        if unmeasured_animal:
            meter_uuid = next((obj["uuid"] for obj in observation["ui"]["inventoryObjects"] if "proteomics meter" in
             \  obj["name"].lower()), None)
            if meter_uuid:
                logger.logMessage("info", f"Attempting to measure {unmeasured_animal['name']}")
                return {"action": "USE", "arg1": meter_uuid, "arg2": unmeasured_animal["uuid"]}

    # If nothing else to do, explore
    return self.get_exploration_action(observation)

def update(self, observation, last_action=None):
    """Update knowledge graph based on new observation"""
    self.step_counter += 1
    self.extract_objects_and_properties(observation)

    if last_action:
        self.update_protein_levels(observation, last_action)

    self.save_graph()

def extract_objects_and_properties(self, observation):
    """Extract objects and their properties from an observation"""
    logger.logMessage("info", f"Extracting objects and properties from observation at step {self.step_counter}")

    # Update inventory and meter status
    self.inventory = set(obj["uuid"] for obj in observation["ui"]["inventoryObjects"])
    self.has_meter = any("proteomics meter" in obj["name"].lower() for obj in observation["ui"]["inventoryObjects"])
    if self.has_meter:
        logger.logMessage("info", "Agent has acquired proteomics meter")

    # Update location tracking
    current_location = (
        observation["ui"]["agentLocation"]["x"],
        observation["ui"]["agentLocation"]["y"]
    )
    if current_location == self.last_location:
        self.stuck_counter += 1
    else:
        self.stuck_counter = 0
        self.visited_locations.add(current_location)
    self.last_location = current_location

    # Extract objects and their properties
    for obj in observation["ui"]["inventoryObjects"] + observation["ui"]["accessibleEnvironmentObjects"]:
        obj_name = obj["name"].lower()
        obj_uuid = obj["uuid"]
        self.object_uuids[obj_name] = obj_uuid

        self.add_node(obj_name, "object")

        if obj["description"]:
            desc_node = f"property_{obj['description']}"
            self.add_node(desc_node, "property")
            self.add_edge(obj_name, desc_node, "has_description")

def update_protein_levels(self, observation, action):
    """Update protein levels based on measurement results"""
    if action["action"] == "USE" and any(obj["uuid"] == action["arg1"] and "proteomics meter" in obj["name"].lower()
     \  for obj in observation["ui"]["inventoryObjects"]):
        target_obj = next((obj for obj in observation["ui"]["accessibleEnvironmentObjects"] if obj["uuid"] ==
         \  action["arg2"]), None)
        if target_obj:
            if "statue" in target_obj["name"].lower():
                self.measured_statues.add(target_obj["uuid"])
                return

            if target_obj["uuid"] not in self.measurement_attempts:
                self.measurement_attempts[target_obj["uuid"]] = 0
```

```python
                self.measurement_attempts[target_obj["uuid"]] += 1

                message = observation["ui"]["lastActionMessage"].lower()
                logger.logMessage("info", f"Attempting to parse protein levels from: {message}")

                try:
                    if "protein a:" in message and "protein b:" in message:
                        protein_a = float(message.split("protein a:")[1].split()[0])
                        protein_b = float(message.split("protein b:")[1].split()[0])

                        self.protein_levels[target_obj["name"]] = {
                            "protein_a": protein_a,
                            "protein_b": protein_b
                        }
                        self.measured_animals.add(target_obj["uuid"])
                        self.successful_measurements.add(target_obj["uuid"])

                        # Add measurement to knowledge graph
                        measurement_node = f"measurement_{target_obj['name']}_{self.step_counter}"
                        self.add_node(measurement_node, "measurement")
                        self.add_edge(target_obj["name"], measurement_node, "measured_at")

                        protein_a_node = f"protein_a_{protein_a:.2f}"
                        protein_b_node = f"protein_b_{protein_b:.2f}"
                        self.add_node(protein_a_node, "property")
                        self.add_node(protein_b_node, "property")
                        self.add_edge(measurement_node, protein_a_node, "protein_a")
                        self.add_edge(measurement_node, protein_b_node, "protein_b")

                        logger.logMessage("info", f"Successfully recorded protein levels for {target_obj['name']}")

                        # Analyze for outliers
                        outliers = self.analyze_protein_levels()
                        if outliers:
                            logger.logMessage("info", f"Identified outliers: {outliers}")

                        self.exploration_steps = 0
                except Exception as e:
                    logger.logMessage("error", f"Failed to parse protein levels: {str(e)}")

def run_episode(agent, api, seed, difficulty, max_steps, agent_type="knowledge_graph"):
    """Run a single episode"""
    logger.logMessage("info", f"Starting episode with seed {seed}, difficulty {difficulty}, agent type {agent_type}")

    success = api.loadScenario("Proteomics", difficulty, seed, 1)
    if not success:
        logger.logMessage("error", "Failed to load scenario")
        return None

    observation = api.getAgentObservation(0)

    last_action = None
    for step in range(max_steps):
        if agent_type == "knowledge_graph":
            agent.update(observation, last_action)
        else:
            agent.update(observation, last_action)

        action = agent.think(observation)

        if action is None:
            logger.logMessage("info", "Agent has completed its task or cannot determine next action")
            break

        logger.logMessage("info", f"Taking action: {json.dumps(action)}")

        result = api.performAgentAction(0, action)
        if not result.get("success", False):
            logger.logMessage("warning", f"Action failed: {json.dumps(result)}")

        last_action = action

        api.tick()
        observation = api.getAgentObservation(0)

        if api.world.taskScorer.tasks[0].completed:
            break

    scorecard = api.getTaskScorecard()[0]

    result = {
        "completion": int(scorecard["completed"]),
        "success": int(scorecard["completedSuccessfully"]),
        "process_score": scorecard["scoreNormalized"],
        "steps": step + 1,
```

```
        "protein_levels": agent.protein_levels
    }

    if agent_type == "knowledge_graph":
        result.update({
            "nodes": len(agent.nodes),
            "edges": len(agent.edges),
            "hypotheses": list(agent.hypotheses)
        })

    return result

def plot_results(results):
    """Generate plots comparing baseline and experimental conditions"""
    # Separate results by agent type
    kg_results = [r for r in results["episodes"] if r["agent_type"] == "knowledge_graph"]
    baseline_results = [r for r in results["episodes"] if r["agent_type"] == "baseline"]

    # Plot process scores
    plt.figure(figsize=(10, 6))
    plt.plot([r["process_score"] for r in kg_results], label="Knowledge Graph Agent")
    plt.plot([r["process_score"] for r in baseline_results], label="Baseline Agent")
    plt.xlabel("Episode")
    plt.ylabel("Process Score")
    plt.title("Process Scores by Agent Type")
    plt.legend()
    plt.savefig("to_save/process_scores.pdf")
    plt.close()

    # Plot graph complexity for knowledge graph agent
    plt.figure(figsize=(10, 6))
    plt.plot([r["nodes"] for r in kg_results], label="Nodes")
    plt.plot([r["edges"] for r in kg_results], label="Edges")
    plt.xlabel("Episode")
    plt.ylabel("Count")
    plt.title("Knowledge Graph Complexity")
    plt.legend()
    plt.savefig("to_save/graph_complexity.pdf")
    plt.close()

def analyze_results(results):
    """Perform statistical analysis of results"""
    kg_scores = [r["process_score"] for r in results["episodes"] if r["agent_type"] == "knowledge_graph"]
    baseline_scores = [r["process_score"] for r in results["episodes"] if r["agent_type"] == "baseline"]

    difference_scores, mean_baseline, mean_experimental = generate_difference_scores_parallel_arrays(baseline_scores,
     \  kg_scores)
    bootstrap_results = bootstrap_resampling(difference_scores, mean_baseline, mean_experimental)

    return bootstrap_results

def main():
    results = {
        "pilot_mode": PILOT_MODE,
        "timestamp": datetime.now().isoformat(),
        "episodes": []
    }

    for difficulty in DIFFICULTIES:
        for seed in SEEDS:
            # Run knowledge graph agent
            kg_agent = KnowledgeGraphAgent(thread_id=seed)
            kg_api = DiscoveryWorldAPI(threadID=seed)
            kg_result = run_episode(kg_agent, kg_api, seed, difficulty, MAX_STEPS_PER_EPISODE, "knowledge_graph")

            if kg_result:
                kg_result.update({
                    "difficulty": difficulty,
                    "seed": seed,
                    "agent_type": "knowledge_graph"
                })
                results["episodes"].append(kg_result)
                logger.logMessage("info", f"Completed knowledge graph agent episode: difficulty={difficulty}, seed={seed},
                 \  success={kg_result['success']}")

            # Run baseline agent
            baseline_agent = BaselineReActAgent(thread_id=seed)
            baseline_api = DiscoveryWorldAPI(threadID=seed)
            baseline_result = run_episode(baseline_agent, baseline_api, seed, difficulty, MAX_STEPS_PER_EPISODE,
             \  "baseline")

            if baseline_result:
                baseline_result.update({
                    "difficulty": difficulty,
                    "seed": seed,
```

```
                    "agent_type": "baseline"
                })
                results["episodes"].append(baseline_result)
                logger.logMessage("info", f"Completed baseline agent episode: difficulty={difficulty}, seed={seed},
                 \  success={baseline_result['success']}")

    # Generate plots
    plot_results(results)

    # Perform statistical analysis
    stats = analyze_results(results)
    results["statistical_analysis"] = stats

    # Save results
    with open("results.json", "w") as f:
        json.dump(results, f, indent=2)

if __name__ == "__main__":
    main()
```

Listing 12: CodeScientist generated code for this experiment.

H.7 Report: Combinatorial Optimization

Evaluation of LLM-Based vs Mathematical Approaches for Resistor Substitution

CodeScientist

February 15th, 2025

Abstract

This paper evaluates three approaches for suggesting resistor combinations to match target resistance values: an LLM-based advisor, a simple baseline using nearest values, and a mathematical optimization approach. Testing on 50 random target values between 10Ω and $1M\Omega$ showed that while the LLM approach achieved only 24% accuracy within 1% tolerance, the simple baseline achieved 94% and the mathematical optimization achieved 100%. The results suggest that for this well-defined numerical problem, traditional algorithmic approaches outperform current LLM capabilities.

1 Introduction

Finding optimal combinations of standard resistor values to match a target resistance is a common electronics design challenge. This study compares three approaches:

- LLM-based: Using GPT-4 to suggest combinations based on natural language prompting
- Simple baseline: Finding nearest single value or basic series combination
- Mathematical optimization: Exhaustive search of series/parallel combinations

The implicit hypothesis was that an LLM could learn to suggest effective resistor combinations by understanding the domain concepts through its training. This was tested against mathematical baselines.

2 Methodology

The experiment tested 50 random resistance targets between 10Ω and 1MΩ, with 3 trials per target. For each target, all three methods suggested combinations using standard E24 series values. Performance metrics included:

- Percentage error from target value
- Success rates at 1%, 5%, and 10% tolerances
- Number of components used
- Computation time

3 Results

3.1 Accuracy

The mathematical optimization approach achieved the highest accuracy, with 100% of suggestions within 1% of target values. The simple baseline achieved 94% within 1% tolerance. The LLM approach performed significantly worse, with only 24% within 1% tolerance.

Method	Within 1%	Within 5%	Within 10%
LLM	24.0%	39.3%	46.7%
Simple	94.0%	98.0%	100.0%
Mathematical	100.0%	100.0%	100.0%

Table 1: Success rates at different tolerance levels

3.2 Computation Time

The simple baseline was fastest (0.004s mean), followed by the LLM approach (1.157s), with mathematical optimization slowest (3.226s). However, the mathematical approach's superior accuracy likely justifies its longer computation time for most applications.

3.3 Error Distribution

The error distribution plots (Figure 1) show that while the mathematical and simple approaches maintained consistently low errors, the LLM approach had much higher variance and several extreme errors, including some complete failures where no valid solution was suggested.

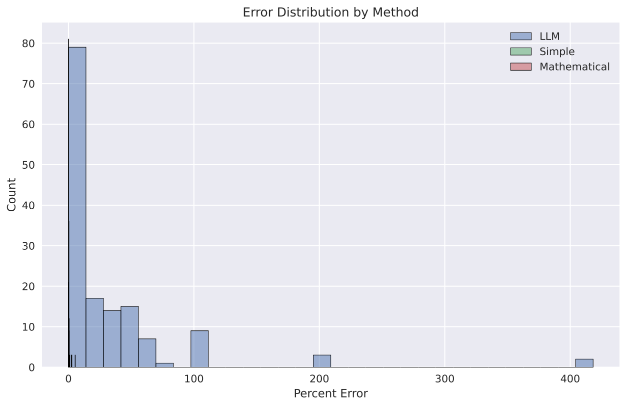

Figure 1: Error distribution by method

4 Discussion

The results clearly reject the hypothesis that the LLM approach could compete with mathematical methods for this task. This is not entirely surprising, as resistor combination optimization is fundamentally a mathematical problem with well-defined rules and constraints.

Key limitations of the LLM approach included:

- Tendency to suggest unnecessarily complex combinations
- Occasional complete failures to generate valid solutions
- Inconsistent performance across different resistance ranges

5 Limitations

The study has several limitations:

- Testing was limited to 50 target values
- Only one LLM model/prompt combination was tested

- Real-world factors like component cost and availability were not considered
- The mathematical optimization's longer computation time might be prohibitive for some applications

6 Conclusion

For resistor substitution advice, traditional algorithmic approaches significantly outperform current LLM capabilities. The simple baseline offers an excellent compromise between accuracy and speed, while mathematical optimization provides maximum accuracy at the cost of computation time. Future work could explore whether improved prompting or different LLM models could achieve better results, but the fundamental mathematical nature of the task suggests traditional algorithms will likely remain superior.

7 (This section added by a human domain expert)

The report generator failed to include one of the figures the experiment generates:

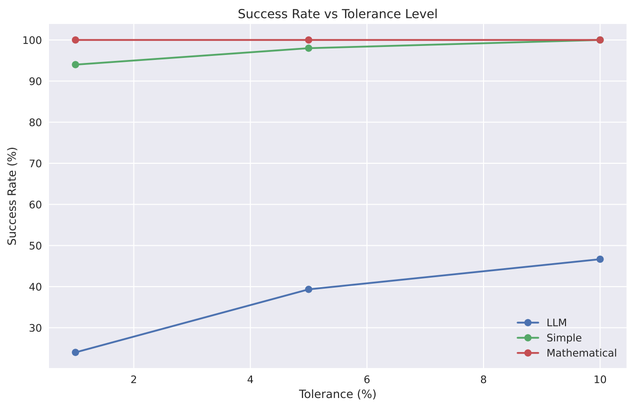

Figure 2: Success Rates

Code Listing: Combinatorial Optimization

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns
import json
import time
import os
from typing import List, Dict, Tuple
import math
from itertools import combinations_with_replacement
from experiment_common_library import Logger, llm_response

# Global configuration
PILOT_MODE = 'PILOT'  # Changed to PILOT mode
logger = Logger()

# Create to_save directory if it doesn't exist
os.makedirs('to_save', exist_ok=True)

class ResistorSeries:
    def __init__(self):
        # E12 series base values
        self.E12 = [10, 12, 15, 18, 22, 27, 33, 39, 47, 56, 68, 82]
        # E24 series adds intermediate values
        self.E24 = [10, 11, 12, 13, 15, 16, 18, 20, 22, 24, 27, 30, 33, 36, 39, 43, 47, 51, 56, 62, 68, 75, 82, 91]

        # Generate full series with multipliers
        self.full_E12 = self._generate_full_series(self.E12)
        self.full_E24 = self._generate_full_series(self.E24)

    def _generate_full_series(self, base_values: List[int]) -> List[float]:
        full_series = []
        for n in range(6):  # 10^0 to 10^5
            multiplier = 10 ** n
            full_series.extend([x * multiplier for x in base_values])
        return sorted(full_series)

def calculate_parallel_resistance(r1: float, r2: float) -> float:
    return (r1 * r2) / (r1 + r2)

def calculate_series_resistance(r1: float, r2: float) -> float:
    return r1 + r2

def calculate_total_resistance(components: List[float], connections: List[str]) -> float:
    if len(components) == 1:
        return components[0]

    current_value = components[0]
    for i in range(1, len(components)):
        if connections[i-1] == 'series':
            current_value = calculate_series_resistance(current_value, components[i])
        else:  # parallel
            current_value = calculate_parallel_resistance(current_value, components[i])
    return current_value

def calculate_percent_error(target: float, actual: float) -> float:
    return abs((actual - target) / target) * 100

class BaselineSimple:
    def __init__(self, available_values: List[float]):
        self.available_values = available_values

    def find_closest_single(self, target: float) -> Dict:
        closest = min(self.available_values, key=lambda x: abs(x - target))
        return {
            'components': [closest],
            'connections': [],
            'actual_resistance': closest
        }

    def find_closest_series(self, target: float) -> Dict:
        best_error = float('inf')
        best_result = None

        for r1 in self.available_values:
            for r2 in self.available_values:
                total = r1 + r2
                error = abs(total - target)
                if error < best_error:
                    best_error = error
                    best_result = {
                        'components': [r1, r2],
                        'connections': ['series'],
```

```
                    'actual_resistance': total
                }

        # Compare with single resistor solution
        single = self.find_closest_single(target)
        if abs(single['actual_resistance'] - target) < best_error:
            return single

        return best_result

class BaselineMathematical:
    def __init__(self, available_values: List[float], max_components: int):
        self.available_values = available_values
        self.max_components = max_components

    def find_optimal_combination(self, target: float) -> Dict:
        best_error = float('inf')
        best_result = None

        # Try different numbers of components
        for n in range(1, self.max_components + 1):
            # Get all possible combinations of n resistors
            for combo in combinations_with_replacement(self.available_values, n):
                # Try different connection patterns
                connection_patterns = self._generate_connection_patterns(n)
                for pattern in connection_patterns:
                    total = calculate_total_resistance(list(combo), pattern)
                    error = abs(total - target)

                    if error < best_error:
                        best_error = error
                        best_result = {
                            'components': list(combo),
                            'connections': pattern,
                            'actual_resistance': total
                        }

        return best_result

    def _generate_connection_patterns(self, n: int) -> List[List[str]]:
        if n <= 1:
            return [[]]
        patterns = []
        for i in range(2 ** (n-1)):
            pattern = []
            for j in range(n-1):
                if (i >> j) & 1:
                    pattern.append('parallel')
                else:
                    pattern.append('series')
            patterns.append(pattern)
        return patterns

class LLMAdvisor:
    def __init__(self, available_values: List[float], max_components: int):
        self.available_values = available_values
        self.max_components = max_components

    def get_suggestion(self, target: float) -> Dict:
        # Enhanced prompt with better guidance about magnitude and parallel combinations
        prompt = f"Given a target resistance of {target} ohms, suggest combinations of up to {self.max_components}
         \  standard resistors from the following series ({[float(x) for x in self.available_values]}) connected in
         \  series and/or parallel to approximate this value.\n\n"
        prompt += "IMPORTANT GUIDELINES:\n"
        prompt += f"1. Choose resistors with values close to the target order of magnitude:\n"
        prompt += f"   - For target {target} ohms, focus on values between {target/10} and {target*10} ohms\n"
        prompt += "   - AVOID using unnecessarily large resistors\n\n"
        prompt += "2. Connection types and formulas:\n"
        prompt += "   - Series: Total = R1 + R2\n"
        prompt += "   - Parallel: Total = 1 / (1/R1 + 1/R2)\n"
        prompt += "   Note: Parallel combinations always result in a total less than the smallest component\n\n"
        prompt += "3. Examples for different magnitudes:\n"
        prompt += '   Target = 150 ohms:\n'
        prompt += '   {"components": [100, 47], "connections": ["series"]}  # 147 ohms\n'
        prompt += '   {"components": [220, 470], "connections": ["parallel"]}  # 151 ohms\n\n'
        prompt += '   Target = 1000 ohms:\n'
        prompt += '   {"components": [680, 330], "connections": ["series"]}  # 1010 ohms\n\n'
        prompt += "Your response must be a valid JSON object with exactly this format:\n"
        prompt += "{\n"
        prompt += '    "components": [value1, value2],  # List of resistance values\n'
        prompt += '    "connections": ["series"]        # List of connection types between adjacent components\n'
        prompt += "}\n\n"
        prompt += "Place your JSON response between triple backticks (```). Do not include any other text."

        logger.logMessage("info", f"Sending prompt to LLM for target resistance: {target} ohms")
```

```
        logger.logMessage("debug", f"Full prompt: {prompt}")

        success, response = llm_response(prompt, "gpt-4o-mini", temperature=0, max_tokens=200)

        if not success:
            logger.logMessage("error", f"LLM call failed: {response}")
            return None

        logger.logMessage("debug", f"Raw LLM response: {response}")

        try:
            # Extract JSON from response using codeblocks
            from experiment_common_library import find_codeblocks
            codeblocks = find_codeblocks(response)

            if not codeblocks:
                logger.logMessage("error", "No codeblocks found in LLM response")
                return None

            # Join the lines and remove any comments
            json_str = '\n'.join(codeblocks[0])
            json_str = '\n'.join([line.split('#')[0].strip() for line in json_str.split('\n')])

            # Validate JSON structure
            suggestion = json.loads(json_str)
            logger.logMessage("debug", f"Parsed suggestion: {suggestion}")

            # Validate the suggestion structure
            if not isinstance(suggestion, dict):
                logger.logMessage("error", "LLM response is not a dictionary")
                return None

            if 'components' not in suggestion or 'connections' not in suggestion:
                logger.logMessage("error", "Missing required keys in LLM response")
                return None

            if not isinstance(suggestion['components'], list) or not isinstance(suggestion['connections'], list):
                logger.logMessage("error", "Components or connections is not a list")
                return None

            if len(suggestion['components']) > 1 and len(suggestion['connections']) != len(suggestion['components']) - 1:
                logger.logMessage("error", "Invalid number of connections for components")
                return None

            # Calculate actual resistance
            actual_resistance = calculate_total_resistance(
                suggestion['components'],
                suggestion['connections']
            )

            suggestion['actual_resistance'] = actual_resistance
            return suggestion

        except Exception as e:
            logger.logMessage("error", f"Error processing LLM response: {str(e)}")
            return None

def plot_results(df: pd.DataFrame, pilot_mode: str):
    plt.style.use('seaborn-v0_8')

    # Error distribution by method
    plt.figure(figsize=(10, 6))
    for method in df['method'].unique():
        method_data = df[df['method'] == method]['percent_error']
        # Filter out infinite values
        method_data = method_data[~np.isinf(method_data)]
        if len(method_data) > 0:  # Only plot if we have valid data
            sns.histplot(data=method_data, bins=30, alpha=0.5, label=method)
    plt.xlabel('Percent Error')
    plt.ylabel('Count')
    plt.title('Error Distribution by Method')
    plt.legend()
    plt.savefig('to_save/error_distribution.pdf')
    plt.close()

    # Success rate vs tolerance level
    tolerances = [1, 5, 10]
    success_rates = []
    for method in df['method'].unique():
        rates = []
        method_data = df[df['method'] == method]
        for tol in tolerances:
            col = f'within_{tol}_percent'
            if col in method_data.columns:
                rate = (method_data[col].mean() * 100)
```

```
                rates.append(float(rate))
        if len(rates) == len(tolerances):  # Only add if we have all tolerance levels
            success_rates.append({'method': method, 'rates': rates})

    if success_rates:  # Only plot if we have valid data
        plt.figure(figsize=(10, 6))
        for data in success_rates:
            plt.plot(tolerances, data['rates'], marker='o', label=data['method'])
        plt.xlabel('Tolerance (%)')
        plt.ylabel('Success Rate (%)')
        plt.title('Success Rate vs Tolerance Level')
        plt.legend()
        plt.savefig('to_save/success_rates.pdf')
        plt.close()

def perform_statistical_analysis(df: pd.DataFrame):
    try:
        from experiment_common_library import generate_difference_scores_dict, bootstrap_resampling

        # Prepare data for bootstrap analysis
        methods = sorted(df['method'].unique())
        baseline_method = 'Simple'

        logger.logMessage("info", f"Performing statistical analysis comparing methods: {methods}")

        for experimental_method in methods:
            if experimental_method == baseline_method:
                continue

            # Prepare data
            comparison_data = []
            for target in df['target_value'].unique():
                baseline_data = df[(df['method'] == baseline_method) &
                                 (df['target_value'] == target)]['percent_error']
                exp_data = df[(df['method'] == experimental_method) &
                             (df['target_value'] == target)]['percent_error']

                if len(baseline_data) > 0 and len(exp_data) > 0:
                    baseline_error = float(baseline_data.iloc[0])
                    exp_error = float(exp_data.iloc[0])

                    # Skip infinite values
                    if not np.isinf(baseline_error) and not np.isinf(exp_error):
                        comparison_data.append({
                            'baseline_score': -baseline_error,  # Negative because lower error is better
                            'experimental_score': -exp_error
                        })

            if len(comparison_data) > 0:
                logger.logMessage("info", f"Comparing {experimental_method} vs {baseline_method} with
                 \ {len(comparison_data)} valid comparison points")

                # Perform bootstrap analysis
                difference_scores, mean_baseline, mean_experimental = generate_difference_scores_dict(
                    comparison_data, 'baseline_score', 'experimental_score'
                )

                results = bootstrap_resampling(difference_scores, mean_baseline, mean_experimental)

                logger.logMessage("info", f"Bootstrap analysis results for {experimental_method} vs {baseline_method}:")
                logger.logMessage("info", json.dumps(results, indent=2))

                # Log interpretation
                p_value = results.get('p_value', 1.0)
                if p_value < 0.05:
                    logger.logMessage("info", f"{experimental_method} is significantly different from {baseline_method} (p
                     \ < 0.05)")
                else:
                    logger.logMessage("info", f"No significant difference between {experimental_method} and
                     \ {baseline_method} (p >= 0.05)")
            else:
                logger.logMessage("warning", f"No valid comparison data for {experimental_method} vs {baseline_method}")

    except Exception as e:
        logger.logMessage("error", f"Error in statistical analysis: {str(e)}")

def run_experiment(pilot_mode: str):
    # Initialize components
    resistor_series = ResistorSeries()

    # Configure experiment based on pilot mode
    if pilot_mode == 'MINI_PILOT':
        num_values = 5
        max_resistors = 2
        available_values = resistor_series.full_E12
```

```
    runs_per_value = 1
elif pilot_mode == 'PILOT':
    num_values = 50  # Changed from 20 to 50 for follow-on experiment
    max_resistors = 3
    available_values = resistor_series.full_E24
    runs_per_value = 3
else:  # FULL_EXPERIMENT
    num_values = 100
    max_resistors = 3
    available_values = resistor_series.full_E24
    runs_per_value = 5

logger.logMessage("info", f"Starting experiment in {pilot_mode} mode with {num_values} values, {max_resistors} max
 \  resistors, and {runs_per_value} runs per value")

# Initialize methods
llm_advisor = LLMAdvisor(available_values, max_resistors)
baseline_simple = BaselineSimple(available_values)
baseline_math = BaselineMathematical(available_values, max_resistors)

# Generate random target values
np.random.seed(42)  # For reproducibility
exp_range = np.log10(np.array([10, 1e6]))  # 10Ω to 1MΩ
target_values = np.power(10, np.random.uniform(exp_range[0], exp_range[1], num_values))

# Initialize results DataFrame
results = []

# Run experiment
for target in target_values:
    logger.logMessage("info", f"Processing target value: {target} ohms")

    for run in range(runs_per_value):
        logger.logMessage("info", f"Run {run + 1}/{runs_per_value}")

        # Test each method
        for method_name, method in [
            ("LLM", llm_advisor),
            ("Simple", baseline_simple),
            ("Mathematical", baseline_math)
        ]:
            start_time = time.time()

            try:
                if method_name == "Simple":
                    suggestion = method.find_closest_series(target)
                elif method_name == "Mathematical":
                    suggestion = method.find_optimal_combination(target)
                else:  # LLM
                    suggestion = method.get_suggestion(target)

                if suggestion is None:
                    logger.logMessage("error", f"Method {method_name} failed for target {target} ohms")
                    # Record failure result
                    result = {
                        'target_value': float(target),
                        'method': method_name,
                        'suggested_components': [],
                        'connection_type': [],
                        'actual_resistance': 0.0,
                        'percent_error': float('inf'),
                        'within_1_percent': False,
                        'within_5_percent': False,
                        'within_10_percent': False,
                        'num_components': 0,
                        'computation_time': float(time.time() - start_time)
                    }
                else:
                    computation_time = time.time() - start_time

                    # Calculate metrics
                    percent_error = calculate_percent_error(target, suggestion['actual_resistance'])

                    # Record results with explicit type conversion
                    result = {
                        'target_value': float(target),
                        'method': method_name,
                        'suggested_components': suggestion['components'],
                        'connection_type': suggestion['connections'],
                        'actual_resistance': float(suggestion['actual_resistance']),
                        'percent_error': float(percent_error),
                        'within_1_percent': bool(percent_error <= 1),
                        'within_5_percent': bool(percent_error <= 5),
                        'within_10_percent': bool(percent_error <= 10),
                        'num_components': int(len(suggestion['components'])),
```

```
                            'computation_time': float(computation_time)
                        }

                    results.append(result)
                    logger.logMessage("debug", f"Result for {method_name}: {json.dumps(result)}")

                except Exception as e:
                    logger.logMessage("error", f"Error running {method_name}: {str(e)}")
                    continue

    # Convert results to DataFrame
    df = pd.DataFrame(results)

    # Save results
    with open('results.json', 'w') as f:
        json.dump(results, f, indent=2)

    # Generate plots
    plot_results(df, pilot_mode)

    # Perform statistical analysis
    perform_statistical_analysis(df)

    # Log summary statistics
    for method in df['method'].unique():
        method_data = df[df['method'] == method]
        logger.logMessage("info", f"\nSummary for {method}:")
        logger.logMessage("info", f"Mean percent error: {method_data['percent_error'].mean():.2f}%")
        logger.logMessage("info", f"Median percent error: {method_data['percent_error'].median():.2f}%")
        logger.logMessage("info", f"Success rates:")
        logger.logMessage("info", f"  Within 1%: {(method_data['within_1_percent'].mean() * 100):.1f}%")
        logger.logMessage("info", f"  Within 5%: {(method_data['within_5_percent'].mean() * 100):.1f}%")
        logger.logMessage("info", f"  Within 10%: {(method_data['within_10_percent'].mean() * 100):.1f}%")
        logger.logMessage("info", f"Mean computation time: {method_data['computation_time'].mean():.3f} seconds")

def main():
    logger.logMessage("info", f"Starting experiment in {PILOT_MODE} mode")
    run_experiment(PILOT_MODE)
    logger.logMessage("info", "Experiment completed")

if __name__ == "__main__":
    main()
```

Listing 13: CodeScientist generated code for this experiment.