The Icosahedron Experiment: LLM Pair Programming

Reuben Brasher

RB and LPX Foundation

Author Note

ORCID ID: 0009-0006-2982-6908

Email: rb@rb-and-lpx-foundation.org

Abstract

This paper presents a case study of eleven sessions in which a single developer used large language models (LLMs), particularly GPT-3.5 and GPT-4, to assist with structured software development tasks. The study examines model performance in generating, testing, and documenting code through interactive, test-oriented workflows. Drawing on domain features from the D20 role-playing system, the sessions explore how prompt design, task complexity, and stepwise refinement influence code quality and reliability. Results suggest that LLMs can support modular development when guided through incremental tasks and used in ways analogous to pair programming or test-driven design. Practices such as schema definition, explicit testing, and conversational decomposition appeared to improve output consistency and functional correctness. The paper proposes a practical framework for working with LLMs in development contexts and discusses broader considerations for software teams, including developer training, productivity, and ethical concerns.

Keywords: Large Language Models, Prompt Engineering, Program Synthesis, Code Generation, Human–AI Collaboration, Structured Prompting, Model-Assisted Programming

Introduction

Large language models (LLMs), such as GPT, support software development most effectively when used in structured, test-driven workflows. Practices like test-driven development (TDD) and pair programming create the kind of modular, iterative environment in which LLMs can contribute reliably.

Because large language models are trained as sequence-to-sequence predictors, they often perform well when translating between structured formats such as tables, JSON objects, and object-oriented class definitions (Vaswani et al., 2017; Radford et al., 2019). In this study, such transformations included stages from natural language to tabular schemas, to JSON, to data models, to unit tests and documentation. Structured prompts—multi-stage instructions that define each transformation—and iterative human review appeared to support more modular and verifiable code. Later sections present case studies in which this scaffolding shaped model behavior and improved output stability.

This paper documents eleven case studies using GPT in the ChatGPT interface. These experiments test the model's ability to operate as a collaborative agent across diverse programming tasks. The findings show how prompt structure and task complexity influence code quality and support a development methodology that treats LLMs not as autonomous generators, but as reasoning partners in stepwise, test-driven workflows.

All code, prompt logs, and generated artifacts are available at: https://github.com/newexo/icosahedron

Background and Methodological Foundations

This section outlines relevant methodologies and model behaviors that inform the experimental design.

Developer Workflows: Testing, Reasoning, and Collaboration

Professional software development relies on iterative methodologies that promote stability through structured refinement. Test-driven development (TDD) exemplifies this approach. Developers begin by writing tests that define expected behavior, then implement the minimal code needed to pass those tests, followed by immediate refactoring and re-testing (Beck, 2003). This practice enforces modular design, catches regressions early, and enables system growth through the incremental addition of testable units.

Developers rarely begin from scratch. They often adapt code snippets from documentation or forums, modifying them to fit existing modules and architectural constraints. This process requires testing, decomposition, and iterative refinement to transform isolated examples into cohesive, functional components.

Pair programming builds on TDD by adding real-time dialogue and shared reasoning. One developer, the “driver,” writes code, while the “navigator” monitors each step, asks questions, and considers broader design implications (Beck & Andres, 2004). This collaboration compels both participants to explain their thinking, justify each decision, and refine implementation strategies. The navigator often prompts early test writing, flags edge cases, and maintains alignment with project goals. In practice, pair programming reinforces modular design, promotes continuous validation, and supports disciplined, iterative progress.

Emergence of Structured Reasoning in LLMs

Transformer-based language models evolved from innovations in attention mechanisms. Bahdanau et al. (2014) introduced dynamic attention for neural machine translation, enabling models to weight input tokens contextually. Vaswani et al. (2017) replaced recurrence with self-attention, allowing parallel token processing and improving sequence-to-sequence performance. These advances enabled models to translate natural language into structured formats such as tables, JSON, and code.

GPT-style models trained on next-token prediction (Radford et al., 2019; Brown et al., 2020) demonstrated in-context learning and few-shot generalization, allowing them to infer task structure from examples. Austin et al. (2021) showed that such models could generate functional Python from textual prompts. Wang et al. (2021) used encoder-decoder architectures like CodeT5 to improve accuracy with task-specific objectives. These studies confirm that LLMs perform best when translating across representational stages.

Chain-of-thought prompting reinforces this structure. Wei et al. (2022) showed that step-by-step examples improve accuracy on reasoning tasks. Kojima et al. (2022) found that simple cues like “Let’s think step by step” elicit intermediate reasoning. Nangia et al. (2022) applied this to code with scratchpads, and Jin et al. (2023) introduced self-revision through the Self-Refine framework.

Terminology

Drawing from educational theory and human-computer interaction, *scaffolding* refers to structured, incremental support that enables users or learners to perform complex tasks they could not complete independently (Wood, Bruner, & Ross, 1976). In the context of this study, scaffolding refers to prompt sequences, representational transformations, and iterative refinement strategies that help large language models generate modular, testable code.

This study distinguishes between several types of scaffolding. *Priming* refers to initial natural-language prompts that frame the domain, define terminology, and establish expectations. *Grounding* introduces structured representations such as tables, field specifications, or formal schemas to constrain the model’s outputs and align them with representational requirements.

Task decomposition operates at a higher level, breaking complex programming challenges into smaller, testable components such as serialization, validation, unit testing, and documentation. These forms of scaffolding serve not merely to direct the model's attention but to stabilize behavior across representational shifts and support consistent performance in iterative workflows.

Methodological Overview

This section outlines the experimental procedures and methodology used to evaluate large language models (LLMs), specifically GPT-3.5 and GPT-4, in structured software development workflows. It explains how the experiments organized tasks, structured interactions with the model, and documented and assessed results.

Case Study Design

From May 2023 to mid-2024, this study examined how large language models, specifically GPT-3.5 and GPT-4o accessed through the ChatGPT interface, contribute to structured software development workflows. Each case study targeted a discrete feature of the D20 3.5 game system and aimed to produce modular, testable code through staged interaction with the model. Each session began with a priming phase, in which the user established task context through natural language prompts that defined domain concepts, constraints, or intended behaviors. This was followed by grounding prompts that introduced structured representations—such as tables, data schemas, or field specifications—to guide the model toward consistent, testable outputs including serializable data models, class definitions, unit tests, and inline documentation.

Experiments varied in complexity and scope, ranging from simple generators and static data objects to models involving relational logic and conditional behavior. The model produced initial code artifacts in response to functional prompts and received follow-up instructions to generate tests, serialize objects, and document class methods. In several cases, prompts introduced modifications or constraints that required the model to revise earlier outputs or adapt previously generated structures to new requirements.

Each experiment produced two structured documentation files. The transcript recorded the complete conversational sequence, including every prompt and response, and preserved the progression of semantic framing, code generation, testing, and revision. The researcher later annotated these transcripts to categorize prompt types, classify error patterns, and track shifts between data formats and representations. The discussion file provided a curated summary of the session. It outlined prompt strategy, error correction, code pruning, and observed model behavior, and described the rationale for any modifications to feature scope or representational structure.

For each case study, the researcher tracked all code in a public version-controlled repository using two branches: one for raw model outputs and another for corrected implementations. Commit histories mirrored the revision process described in the discussion files and served as a detailed record of how each software component evolved from prompt to testable code.

Documentation and Case Review Procedures

This study documents a series of interactive programming sessions in which a single developer used large language models (LLMs), specifically GPT-3.5 and GPT-4 as accessed through the ChatGPT interface, to perform structured coding tasks. The purpose was not to evaluate model effectiveness in a formal or quantitative sense, but to observe how prompt structure, task complexity, and dialogue sequencing influenced model behavior and output quality.These case studies illustrate patterns of interaction and common challenges encountered when using LLMs in structured coding tasks.

Each case study generated two primary records: a transcript and a discussion file. Transcripts were constructed directly from the ChatGPT session history and organized using a consistent template. They included prompt-response sequences labeled according to their functional role in the interaction: priming, initial code generation, revision requests, and documentation prompts. The corresponding discussion file summarized the coder's reflections on task scope, prompt effectiveness, correction strategy, and the progression from raw output to working implementation. Each experiment used two version-controlled branches to preserve both the original model-generated code and the final, corrected version for each session.

The study did not record the exact model version used in each session. During the early stages of the project, only one model was available through the ChatGPT interface. As the platform evolved, model updates occurred without clear notice, and session metadata did not indicate which version was active. In retrospect, this omission limits reproducibility and interpretability across the full experimental set.

The analysis was qualitative and exploratory. While each experiment followed a broadly consistent workflow, the exact structure varied depending on task complexity and model response. The researcher reviewed each transcript and discussion file individually, noting recurring patterns such as prompt clarification, breakdowns in format or structure alignment, and strategies used to correct model errors. Appendix B presents summary tables for each case study, including descriptive statistics such as prompt count and revision frequency. However, no systematic comparison across experiments was attempted, and these counts should not be interpreted as evaluative metrics.

This documentation framework provided a structured foundation for the case analyses discussed in the Results section. It also surfaced areas where further methodological development—such as formal annotation of prompt types or error categories—may be necessary to support generalizable claims in future work.

Results

This section focuses on three case studies selected from a broader set of eleven experiments documented in Appendix B. These cases were chosen not to represent the full range of tasks, but to exemplify key methodological patterns and developmental insights that emerged over time. The first case illustrates difficulties with unstructured logic generation, the second highlights failure modes in hierarchical modeling, and the third demonstrates a successful application of scaffolded prompt design. Taken together, they show how prompt structure and task decomposition influenced the accuracy, coherence, and testability of model outputs across the study.

Case 1: Procedural Randomness in Ability Score Generation

In tabletop role-playing games that follow the D20 system, each character is assigned six numerical attributes—Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma—that affect actions such as combat, skill checks, and spellcasting. These scores are typically generated using predefined randomization methods. The most common variants involve rolling dice: the “classic” method rolls three six-sided dice and sums them; the “standard” method rolls four dice, drops the lowest, and sums the rest; and the “heroic” method adds a bonus to one of the resulting values.

The user began the session with prompts describing these rules in natural language. The model responded with accurate summaries of the system and its mechanics. Because the user provided no structured input or schema, the model transitioned to code generation without constraints or formal representations.

The user requested an implementation of the standard method. The model produced a functionally correct version, but its use of nondeterministic randomization prevented reproducible output. When asked to revise the code to support consistent output, the model incorporated a seeded random number generator. The user then prompted the model to generalize the design for multiple methods. The model reorganized the implementation to separate shared logic from method-specific behavior.

When asked to implement the classic and heroic variants, the model preserved the structural pattern but introduced logical errors. It incorrectly treated the classic method as a uniform random draw between 3 and 18, rather than a sum of dice rolls. It also omitted the bonus from the heroic method. These mistakes suggest that the model retained syntactic structure but failed to reason about probability or apply domain knowledge introduced earlier.

The user then requested tests. The model produced simple checks to verify that outputs fell within an expected range. When asked for a regression test using a fixed random seed, the model fabricated a plausible result rather than computing it based on actual output. It introduced no additional validation or test reuse without explicit instruction.

The model reproduced structural patterns and implemented user-directed revisions accurately. However, it failed to reason about probability distributions or validate its own outputs. For example, it substituted a uniform random integer for a multi-dice sum and fabricated regression test results without empirical grounding. These failures followed a prompt sequence that lacked formal scaffolding, suggesting that the absence of structured input limited the model's capacity for reliable reasoning.

Case 2: Inventory Items — Failure Modes in Hierarchical Code Generation

In D20-based tabletop role-playing games, characters acquire and use various items, including weapons, armor, tools, and magical artifacts. Although these appear in flat lists on paper character sheets, their structural differences require hierarchical representations in code. For example, weapons include attributes such as damage type and range, while armor specifies defense bonuses and encumbrance.

The session began with a general discussion of item categories. The model described typical features for weapons, armor, and magical objects with broad accuracy. The user then requested structured examples using JSON (JavaScript Object Notation), a common format for representing nested data. The model returned plausible examples, but field names, optional attributes, and data structures varied inconsistently across categories. No schema or shared vocabulary guided the model's output, and the conversation lacked constraints to enforce alignment across item types.

Next, the user asked the model to generate Python code to implement these item types. The model produced a hierarchy of classes—one general base class and several category-specific subclasses. However, the implementation failed to match the earlier JSON examples. Constructor signatures were inconsistent; some attributes were duplicated across classes, while others lacked clear typing or were omitted entirely. It also misapplied basic object-oriented techniques, such as inheritance, resulting in ambiguous and inflexible designs.

When asked to generate tests, the model created examples that failed to instantiate the classes it had defined. Field mismatches and inconsistent parameter ordering caused test failures, which the user corrected manually. The model did not include checks for invalid input, edge cases, or reusable test logic. It offered no strategy for organizing or extending the test suite beyond the immediate prompt.

During refinement, the model fixed individual errors when prompted but did not generalize from earlier corrections or revise its abstractions. It did not restructure flawed hierarchies or apply previous fixes to related components. The session concluded without a documentation phase, and the model did not suggest comments, usage guidance, or explanatory examples on its own.

This case study illustrates the model's limited ability to manage hierarchical or polymorphic data structures without strong scaffolding. The model performed well on isolated tasks but failed to preserve consistency or reuse across representations. Without a unified schema or validation mechanism like Pydantic, it could not maintain semantic alignment across data, code, and tests.

Case 3: Status Effects — High Accuracy Through Layered Representations

In D20-based tabletop role-playing games, status effects represent conditions such as "poisoned," "blinded," or "invisible" that temporarily alter a character's abilities. These effects influence movement, attacks, and other in-game actions and are usually defined by properties such as duration, resistance type, and means of removal.

The user began by asking the model to describe a wide range of status effects. The model responded with accurate summaries, and the user then prompted it to compile this information into a table. The model generated a structured representation with five consistent fields: name, description, duration, saving throw or check, and cure or removal method. This stable schema provided a foundation for subsequent code generation.

The model then converted the table into a structured data object using JSON (a format for encoding key-value pairs). When prompted to create a formal data model, it used Pydantic, a Python library that enforces type constraints and validates structured data. The resulting code accurately reflected the schema with no errors. The user then asked for tests using pytest, a Python tool for writing and running tests. The model produced valid checks for object creation, data parsing, and data serialization.

No corrections or refinements were necessary. The model followed instructions precisely and maintained alignment between schema, code, and tests. Although it did not generate documentation or inline comments, the dialogue had already clarified the structure and semantics.

This case showed that when given a well-defined schema and clear task decomposition, the model could generate accurate and coherent outputs without further guidance. The structured prompts provided a scaffold that constrained the model's responses and improved reliability.

Cross-Case Analysis

The three case studies illustrate how prompt structure, representational scaffolding, and task complexity shaped model performance across the full set of eleven cases. In the absence of structured input, as seen in the ability score generator, the model produced syntactically correct but semantically unreliable code. In the inventory item task, the model struggled with consistency and reuse when asked to produce hierarchical representations without schema guidance. By contrast, the status effects task showed that layered prompts with fixed field structures supported accurate code generation, test construction, and schema alignment. These patterns recur throughout the remaining case studies and underscore the importance of prompt design and task decomposition when working with large language models in development workflows. Appendix B provides summaries of each case, including prompt sequences, corrections, and test outcomes.

Discussion

Prompt design, task decomposition, and representational structure significantly influenced the stability and reliability of model-generated code. Effective outputs aligned with established software engineering practices when instructions proceeded in clearly defined stages, supported by structured inputs and immediate validation. These factors also shaped the extent to which large language models could maintain coherence across iterations and adapt to revision.

Practical Patterns for Effective LLM Use

LLMs contribute most effectively when integrated into workflows that follow test-driven development (TDD). As demonstrated in the Ability Score Roller and Status Effects experiments, developers achieve better outcomes by decomposing tasks into small, well-defined steps and prompting the model to complete and verify each component incrementally. This process improves both functional correctness and modularity.

Treating the model as a collaborator in a pair programming setting further improves reliability. When prompted with examples, asked to explain its reasoning, and tested immediately, the model behaves like a junior developer—capable but in need of supervision. Developers may find that working with LLMs in a pair programming mode reduces cognitive load and streamlines debugging, particularly when prompt sequences encourage immediate feedback and structured reflection.

Implications for Engineering Management

For engineering managers, LLMs can yield productivity gains when workflows emphasize structure, feedback, and oversight. In the experiments presented here—particularly those involving status effects and skills—model performance improved when prompt scaffolding, testing infrastructure, and review mechanisms were in place. These practices align with established engineering norms and appear to support more consistent and reliable outputs. Rather than replacing developers, LLMs augment experienced programmers by automating boilerplate tasks and supporting modular, testable implementations.

Teams that develop prompt templates, enforce documentation standards, and provide LLM-specific onboarding are more likely to benefit from model integration. By contrast, cases characterized by loosely structured interaction—such as the Inventory Item experiment—often produced brittle or inconsistent code.

Implications for Developer Productivity and Upskilling

The experiments suggest that large language models (LLMs) can enhance developer productivity, particularly when navigating unfamiliar languages or frameworks. In sessions where prompt structure was clear and tasks were decomposed into discrete components—such as the Status Effects experiment—models performed reliably with minimal intervention. By contrast, more complex or underspecified tasks, such as the Inventory Item case, required repeated correction and manual restructuring. These outcomes indicate that effective use depends on metacognitive skills: users must anticipate model limitations, frame precise prompts, and critically assess outputs.

The findings also highlight the limitations of claims that LLMs can replace professional programmers. While models can automate routine tasks such as boilerplate generation or documentation, they do not autonomously produce robust or maintainable systems. Novice users may struggle to identify subtle errors or gaps in model output, whereas experienced developers benefit more consistently from structured collaboration.

Finally, the study points to a need for revised training paradigms. Developers must not only understand programming languages and design principles but also learn to interact productively with probabilistic, non-deterministic systems. These results position LLMs not as substitutes for expertise but as tools that, when embedded in test-driven workflows, extend and accelerate human capability in software development.

Ethical and Socioeconomic Considerations

This study selected the D20 system as its domain in part due to the permissive Open Game License (OGL v1.0a), which explicitly allows derivative works and redistribution of System Reference Document (SRD) content (Wizards of the Coast, 2000b). This decision ensured that experiments involving code synthesis and transformation were grounded in a well-documented system with few licensing restrictions. The SRD functions similarly to open-source or public domain materials, where derivative use is explicitly permitted under defined conditions. In this context, even if the model inadvertently reproduced fragments of training data resembling SRD content, the resulting outputs would likely qualify either as fair use or as permitted derivatives under the license. The intention here is technical rather than legal: the use of open, permissively licensed materials helped minimize the risk of ambiguous reuse and supported reproducibility.

Nonetheless, broader ethical concerns about intellectual property and data provenance remain. Developers must carefully evaluate LLM-generated code for unintended reproduction of proprietary or licensed material—particularly where model training data may include examples from open repositories with varied or unclear licensing terms. Transparent workflows, including preserved transcripts and version-controlled outputs, help mitigate these risks by enabling traceability and review.

Beyond these technical and procedural safeguards, the integration of LLMs into software development also raises socioeconomic questions about how such tools may reshape professional roles and training pathways.

The experiments suggest that LLMs can increase productivity for experienced developers, particularly in tasks involving boilerplate generation, schema transformation, or test writing. However, they do not eliminate the need for expert oversight and debugging. This asymmetry in utility may contribute to a polarized labor structure in which senior developers supervise model-assisted workflows, while opportunities for novice developers to gain early experience may become more limited. Further investigation is needed to assess how model integration reshapes access to entry-level roles and long-term skill development.

Future Research

This study raises several questions that future research could pursue with greater methodological rigor. The experiments reported here were exploratory and conducted by a single researcher without formal control groups, standardized tasks, or consistent model versioning in early phases. A follow-up study could establish a curated task set, assign tasks to multiple developers of varying experience levels, and track performance using predefined metrics such as code correctness, number of revisions, and test coverage. By adopting a structured evaluation framework, such a study could provide stronger empirical support for claims about the effectiveness of prompt scaffolding and model-assisted workflows.

Another promising direction involves the design of collaborative workflows that assign specific roles to human and model participants. This paper draws on the concept of pair programming, but future studies might formalize this structure by dividing labor between “driver” and “navigator” roles, or between higher-level planning and lower-level implementation. Such divisions could also be applied to agentic systems, where multiple LLM instances perform complementary roles—such as schema designer, code generator, and test validator—within a shared task environment. Comparing these multi-agent architectures to human–LLM interaction may reveal how role differentiation and prompt design interact to support reliable, scalable development.

Finally, the insights from this case study could inform larger conversations about programmer productivity, especially when linked to psychological studies of software engineering practice. Existing literature in human-computer interaction, cognition, and developer tooling could provide theoretical foundations for evaluating LLM-supported workflows more systematically. In particular, connecting task decomposition and prompt scaffolding to known principles of cognitive load, attention management, and error detection may yield frameworks for optimizing LLM integration into real-world development environments.

Conclusion

This study examined how large language models can contribute to structured software development when guided through prompt scaffolding, task decomposition, and iterative validation. Across a set of exploratory case studies, models produced the most reliable and coherent outputs when treated as reasoning assistants rather than autonomous agents. Prompt sequences that introduced structured schemas, enforced representational alignment, and elicited testable components supported greater modularity and functional accuracy.

The documented methodology—including preserved transcripts, annotated discussion files, and version-controlled code histories—provides a foundation for reproducible analysis and future experimentation. While the study relied on a single developer and lacked formal controls, its findings indicate that established practices such as test-driven development and pair programming can be adapted to support productive collaboration between humans and generative models.

As LLMs become increasingly integrated into software workflows, the need grows for rigorous methods to evaluate, guide, and constrain their use. These models do not eliminate the need for expertise; rather, they extend the capabilities of skilled practitioners when embedded in structured environments. Future research should continue to refine the interaction patterns, evaluation criteria, and collaborative roles that enable reliable, scalable human–AI programming.

References

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., & Sutton, C. (2021). *Program synthesis with large language models*. arXiv. https://arxiv.org/abs/2108.07732

Bahdanau, D., Cho, K., & Bengio, Y. (2014). *Neural machine translation by jointly learning to align and translate*. arXiv. https://arxiv.org/abs/1409.0473

Beck, K., & Andres, C. (2004). *Extreme programming explained : embrace change* (2nd ed.). Addison-Wesley.

Beck, K. (2003). *Test-driven development : by example* (1st edition). Addison-Wesley.

Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J. D., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., ... Amodei, D. (2020). *Language models are few-shot learners*. arXiv. https://arxiv.org/abs/2005.14165

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. de O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., Ray, A., Puri, R., Krueger, G., Petrov, M., Khlaaf, H., Sastry, G., Mishkin, P., Chan, B., Gray, S., … Zaremba, W. (2021). *Evaluating Large Language Models Trained on Code*. https://doi.org/10.48550/arxiv.2107.03374

Jin, W., Wang, X., Santurkar, S., Wang, X., Le, Q. V., Jaakkola, T., & Zhou, D. (2023). *Self-Refine: Iterative refinement with self-feedback*. arXiv. https://arxiv.org/abs/2303.17651

Kojima, T., Gu, S. S., Reid, M., Matsuo, Y., & Iwasawa, Y. (2022). *Large language models are zero-shot reasoners*. arXiv. https://arxiv.org/abs/2205.11916

Nangia, N., Phang, J., Bordia, S., Dhingra, B., & Bowman, S. R. (2022). *Show your work: Scratchpads for intermediate computation with language models*. arXiv. https://arxiv.org/abs/2202.00417

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., & Sutskever, I. (2019). *Language models are unsupervised multitask learners* (OpenAI Technical Report). https://cdn.openai.com/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., & Polosukhin, I. (2017). *Attention is all you need*. In *Advances in Neural Information Processing Systems* (NeurIPS 2017). arXiv. https://arxiv.org/abs/1706.03762

Wang, Y., Wang, W., Joty, S., & Hoi, S. C. H. (2021). *CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation*. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing (EMNLP 2021)*. Association for Computational Linguistics. https://arxiv.org/abs/2109.00859

Wang, X., Wei, J., Schuurmans, D., Le, Q. V., Chi, E. H., Narang, S., Chowdhery, A., & Zhou, D. (2022). *Self-consistency improves chain of thought reasoning in language models*. arXiv. https://arxiv.org/abs/2203.11171

Wei, J., Wang, X., Schuurmans, D., Bosma, M., Chi, E., Le, Q., & Zhou, D. (2022). *Chain of thought prompting elicits reasoning in large language models*. arXiv. https://arxiv.org/abs/2201.11903

Wizards of the Coast. (2000a). *Dungeons & Dragons 3.5 System Reference Document (SRD)* [RTF files]. Internet Archive. https://archive.org/details/dnd35srd

Wizards of the Coast. (2000b). *Open Game License v1.0a and legal information from the System Reference Document*. [Legal.rtf]. Wizards of the Coast, Inc. https://archive.org/download/dnd35srd/Legal.rtf

Wood, D., Bruner, J. S., & Ross, G. (1976). The role of tutoring in problem solving. *Journal of Child Psychology and Psychiatry*, *17*(2), 89–100. https://doi.org/10.1111/j.1469-7610.1976.tb00381.x

Wu, Y., Schuster, M., Chen, Z., Le, Q. V., Norouzi, M., Macherey, W., Krikun, M., Cao, Y., Gao, Q., Macherey, K., Klingner, J., Shah, A., Johnson, M., Liu, X., Kaiser, Ł., Gouws, S., Kato, Y., Kudo, T., Kazawa, H., ... Dean, J. (2016). *Google's neural machine translation system: Bridging the gap between human and machine translation*. arXiv. https://arxiv.org/abs/1609.08144

Appendix A: Overview of the D20 System

The D20 System, as codified in the *Dungeons & Dragons 3.5 Edition System Reference Document* (SRD), is a rules framework that underpins character creation, combat resolution, and skill use in tabletop role-playing games. At its core, it uses a 20-sided die (d20) to resolve actions, where a player rolls a d20, adds relevant modifiers, and compares the result to a difficulty class (DC) or opposing roll.

Characters are defined by six core ability scores: Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma. These scores influence combat, skill checks, and saving throws. For example, a Dexterity modifier might apply to ranged attacks and Armor Class (AC), while Constitution affects hit points.

Game flow is structured around encounters—typically combat or roleplaying scenarios—designed by the Game Master (GM), who also adjudicates rules and controls non-player characters (NPCs). Characters gain experience points (XP) and levels, selecting from base classes (e.g., Fighter, Wizard) and optionally from prestige classes, which grant specialized abilities once prerequisites are met.

Customization is a major feature. The system supports multiclassing, allowing characters to combine levels from different classes to gain hybrid capabilities. Skills, feats, and spells provide further depth, enabling tailored character builds. Variants and optional rules can be used for flexibility, depending on the campaign's tone or complexity.

This ruleset supports structured code representations, as each game element—be it a spell, item, or class—can be expressed through tabular attributes (e.g., AC, damage, spell level), which makes it well suited to LLM-assisted translation into structured formats like JSON and object-oriented code (Wizards of the Coast, 2000).

Appendix B: Full Experiment List

A list of the original transcript and discussion files for these case studies is available at https://github.com/newexo/icosahedron/blob/main/prompts/README.md

Table 1. Summary of Structured Prompting Experiments Across D20 System Features.

Feature	Description	Prompt Rounds	Revisions	Errors Logged	Documentation	Tabular Grounding
Ability Score Roller	Rolling ability scores using standard, classic, and heroic rules	14	4	5	Y	N
Dice Roller	Dice roller for arbitrary die expressions with modifiers	22	4	8	Y	N
Mob Stat Block	NPC data model for encounters and stat resolution	18	3	5	Y	Y
Character Sheet	Character record supporting race, class, and leveling	12	3	4	Y	N
Inventory Item	Item structure with name, type, and value fields	30	5	8	N	N
Skills	Skill mechanics with ability modifiers and class bonuses	16	2	2	Y	Y
Feats	Feat representation including prerequisites and stacking	10	1	1	Y	Y
Spells	Spell definitions with level, school, and casting time	15	3	3	Y	Y
Action	Combat action types and attack-of-opportunity triggers	35	1	1	Y	Y
Character Class	Character class logic for HP, BAB, and skill progression	17	8	5	Y	N
Status Effect	Status condition taxonomy with durations and saving throws	25	1	1	Y	Y

Ability Score Roller

In the D20 system, characters begin with six core attributes—Strength, Dexterity, Constitution, Intelligence, Wisdom, and Charisma—each determined by dice rolls. This experiment modeled three standard generation methods: the *classic* approach (3d6), the *standard* variant (4d6, drop the lowest), and a *heroic* variant that adds +2 to one of the resulting scores. These methods combine randomness with rule-based constraints, offering a structured domain for testing procedural logic and controlled stochastic behavior.

Prompt Structure and Evolution

The session began with natural language prompts describing the three generation rules. From the outset, the design followed an object-oriented pattern, introducing a shared abstract base class with concrete subclasses for each method. No tabular scaffolding or schema-based representations were used. Unit tests were introduced after initial code generation to verify behavior and clarify expectations; they were not prompted by repeated failure but were a standard follow-up practice across sessions. Docstrings were added at the end of the session, as was typical in this experiment series. While the interface design supported modularity, no reuse across other experiments is documented.

Failure Modes and Debugging

Initial outputs included logic errors in the dice aggregation process, particularly in implementing the 4d6-drop-lowest rule, which required multiple revisions. Random number generation also posed challenges: the model could not accurately predict the outcome of seeded RNG sequences, and some early outputs failed basic bounds checks. Additional issues included redundant code, incorrect imports, and unclear class naming. These were resolved by refining method logic, centralizing random state control, and enforcing a cleaner interface structure.

Task Complexity

This experiment combined probabilistic reasoning, object-oriented design, and post-hoc test verification. The interplay of subclass logic, abstract interfaces, and deterministic testing frameworks rendered the task moderately complex. Success was achieved through structured revisions and consistent prompt scaffolding.

Dice Roller

In the D20 system, dice determine outcomes for actions, attacks, and skill checks. A roll is defined by the die type (e.g., d6, d20), the number of dice, and an optional modifier. A reusable and configurable roller is essential for automation in character management and combat resolution.

Prompt Structure and Evolution

The session began with natural language prompts describing rolling mechanics, without use of tabular or schema-based structures. From the outset, the session was scaffolded to include unit testing and JSON serialization. The model initially produced a monolithic class structure, which was later refactored to separate data representation from execution logic.

Failure Modes and Debugging

Initial outputs exhibited a range of implementation errors, including incorrect attribute names, missing constructor arguments, and unusable test cases. Corrections were introduced through user clarification and prompt refinement. Refactoring steps included input validation, centralization of seeded randomness, and the separation of concerns through distinct classes for roll configuration and evaluation. These changes improved modularity, though later work suggested that using an external framework such as Pydantic might have yielded better long-term structure.

Task Complexity

The task required simulating random output with configurable inputs and maintaining reproducibility for testing. Although inheritance was not used, the code involved serialization, validation, and procedural logic. The experiment showed that even modest tasks required stepwise clarification to produce reliable results.

Mob Stat Block

During D20 game play, the dungeon master (DM) running the game will track the details of non-player characters (NPCs) such as enemies or allies using stat blocks. These include hit points, abilities, actions, and descriptive traits used to govern combat and roleplay.

Prompt and Generated Content Evolution

This experiment began with grounding prompts that led directly to structured tabular NPC data. The tabular format transitioned smoothly into JSON and then into class definitions. During the code development stage, the user introduced a shared base class to support reusable serialization methods. Unit tests were implemented without requiring extensive correction.

Failure Modes and Debugging

Few errors occurred. Fixes addressed naming inconsistencies, redundant code, and missing helper methods. A structural refactor introduced the Dictable superclass to consolidate shared logic across experiments.

Task Complexity

This task involved flat data modeling without inheritance or nested representations. Its low complexity and strong schema alignment resulted in minimal revision and high initial output quality. The experiment serves as a clear example of effective code generation for declarative data structures.

Character Sheet

In the D20 system, a character sheet contains all essential information about a player character, including name, class, level, abilities, spells, equipment, and narrative elements such as alignment and backstory.

Prompt and Generated Content Evolution

The initial prompts grounded the task through narrative descriptions of character sheets but failed due to output length limits and incomplete test coverage during the code generation stage. The user responded by explicitly requesting JSON schemas and class definitions with bidirectional methods and validation logic. Unit tests were only successful after further clarification of field names and constructor behavior.

Failure Modes and Debugging

Failures included test exceptions from attribute mismatches, incomplete test coverage, and truncated outputs due to response length. Fixes involved adjusting field names, expanding assertions, and separating prompts into smaller parts. These failures arose from structural mismatches rather than syntactic errors.

Task Complexity

This task involved medium to high complexity, with multiple nested structures and heterogeneous content types. Successful output required decomposing prompts into separate steps and iteratively refining model behavior. The integration of narrative and mechanical components further increased representational demands.

Inventory Item

In the D20 system, inventory items include generic objects (e.g., torches), weapons, armor, and magic rings. Each subclass contains distinct attributes such as damage type, armor bonus, or magical properties.

Prompt Structure and Evolution

This experiment departed from the usual progression by starting with a monolithic prompt that combined example JSON, class definitions, and unit tests. This approach led to failures during initial code development, especially in managing class hierarchies and field alignment. Unlike earlier experiments, there was no refinement of tabular data or modular prompt evolution. Follow-up prompts addressed failures reactively for specific item types but did not establish a reusable scaffold. Although the workflow included unit testing, the absence of structured grounding and the complexity introduced by polymorphism limited generalizability. No documentation phase was performed.

Failure Modes and Debugging

Semantic mismatches, field ordering errors, invalid test scaffolds, and misuse of super() calls caused repeated failures. Debugging required manual intervention to correct constructor logic, align schema assumptions, and remove irrelevant or redundant code. These failures stemmed from the model's difficulty in reasoning over class hierarchies and inherited attributes.

Task Complexity

This was a high-complexity task involving polymorphic class design, schema translation, and multi-level validation. The combination of inheritance, overridden fields, and item-specific logic led to brittle outputs that required extensive human correction. The experiment demonstrates how class hierarchy increases the risk of semantic misalignment and complicates test generation.

Skills

In the D20 system, skills represent a character's learned capabilities in areas such as Acrobatics, Spellcraft, or Stealth. Each skill includes a name, key ability, modifiers, and possibly a trained-only flag.

Prompt Structure and Evolution

This experiment followed the standard progression without deviation. It began with a JSON schema for a skill, followed by prompts for class implementation, unit testing, and documentation. Each stage—initial code development, testing, and docstring generation—proceeded cleanly without error correction or prompt restructuring. The clarity of the schema and flat data structure contributed to high model performance. The experiment exemplifies a minimal-intervention case where prompt patterns generalized directly from earlier low-complexity tasks.

Failure Modes and Debugging

No significant semantic or structural errors occurred. The experimenter requested only a minor revision to one unit test. There were no naming mismatches, serialization issues, or constructor errors.

Task Complexity

This was a low-complexity task. It involved no inheritance, polymorphism, or procedural logic. The schema was flat, and the resulting class had a direct one-to-one mapping with its fields. These conditions made the task well-suited to the model's strengths and minimized the need for human intervention.

Feats

In the D20 system, feats grant characters specific advantages, such as enhanced combat skills or specialized abilities. Each feat includes a name, prerequisites, benefits, and descriptive fields outlining how the ability modifies standard behavior. These are static, schema-defined entries often listed in rulebooks or character builders.

Prompt Structure and Evolution

This experiment began with general natural language discussion of feats and proceeded through the typical schema-driven progression: JSON, Python class, unit tests, and documentation. At the initial code development stage, minor adjustments were required to correct naming mismatches between camelCase in the JSON and snake_case in Python. The flat schema and absence of nested logic supported clean transitions and high generalizability to similar data modeling tasks.

Failure Modes and Debugging

Only one semantic error occurred due to mismatched naming between the JSON schema and the class attribute. There were no syntax or logic errors. This reflects a common issue in LLM outputs when transitioning between formats with different naming conventions.

Task Complexity

This was a low-complexity task. The feat structure included no nesting, no inheritance, and no control flow logic. Prompted transformations were all one-to-one mappings. The simplicity of the schema and the absence of procedural logic enabled near-complete success on the first generation.

Spells

In the D20 system, spells define discrete magical effects available to characters with appropriate class levels and abilities. Each spell includes attributes such as name, level, casting time, duration, components, and range.

Prompt Structure and Evolution

This experiment followed the typical progression from tabular data to JSON, Python class, unit tests, and documentation. During the transition from JSON to code, the user introduced a Dictable base class to consolidate serialization methods, reusing scaffolds from earlier experiments. Minor revisions were required during test generation to correct mismatches between camelCase field names and Python naming conventions, as well as to exclude extraneous fields from the JSON. These corrections were limited in scope and did not require structural reorganization, allowing the schema-first pipeline to proceed with minimal deviation.

Failure Modes and Debugging

Three primary issues occurred: import errors in test files, field name mismatches between JSON and class attributes, and missing or extra fields across formats. These were resolved by correcting import paths, aligning naming conventions, and editing test inputs for schema consistency. All errors were semantic rather than syntactic.

Task Complexity

The task was of moderate complexity. It involved flat but wide schemas, minor field variability, and explicit serialization logic. While no inheritance or control flow was required, successful deserialization depended on consistent attribute alignment. The modularity of the spell data made the structure amenable to LLM output once corrected for naming and schema completeness.

Character Class

In the D20 system, a character’s class determines core progression mechanics such as hit dice, base attack bonus, and skill points per level. These parameters define how characters develop over time and vary by class type (e.g., Fighter, Wizard).

Prompt Structure and Evolution

This experiment began with natural language prompts about character class progression and moved directly to JSON schema construction. From there, prompts guided the generation of a CharacterClass definition and a Character class that computed derived statistics. Minor corrections followed, including clarification of naming conventions, elimination of unused fields, and simplification of computation logic. The pipeline proceeded through the standard JSON to class to unit test sequence. The model applied consistent formulas across multiple class types, and the prompt structure exhibited features that could support reuse in similar tasks.

Failure Modes and Debugging

No runtime errors occurred. The main interventions addressed semantic concerns, including naming inconsistencies, simplification of unused fields, and clarification of formula logic. The model’s initial assumptions about class complexity required user correction but no substantial restructuring.

Task Complexity

This was a low- to medium-complexity task focused on deterministic computation and fixed-structure data modeling. It did not require inheritance or deep nesting, and model outputs stabilized quickly with clear prompts.

Action

The D20 system defines a variety of combat actions—such as casting spells, drawing weapons, or charging an enemy—each associated with specific timing rules and tactical consequences. These actions are categorized by type (Standard, Move, Full-Round, etc.) and vary in whether they provoke attacks of opportunity.

Prompt Structure and Evolution

This experiment began with natural language prompts requesting a list of common D20 actions. The user guided the model through tabular formatting, followed by structured classification and schema design. Multiple iterations refined the action categories and extended edge cases. Prompts then progressed through JSON construction, class implementation using Pydantic, and unit test generation. Serialization methods (to_dict, from_dict) followed established patterns from prior experiments. The structure required no corrections during testing and generalized effectively to other flat, taxonomic datasets.

Failure Modes and Debugging

No runtime or syntax errors occurred. Minor limitations emerged when the model omitted some action categories or failed to recall less common mechanics. These were resolved through user follow-up prompts. Corrections involved content expansion rather than debugging.

Task Complexity

This was a low-complexity task. It involved static data modeling, categorical labeling, and list management. No inheritance or control flow was needed. Despite the breadth of the domain, the structure was flat and predictable, and the model responded well to incremental clarification.

Status Effects

In the D20 system, status effects alter a creature's abilities, behavior, or defenses. Conditions such as "Paralyzed," "Blessed," or "Invisible" affect combat, movement, or saving throws. These effects are often short-term and may include fields like duration, saving throw type, or specific cures.

Prompt Structure and Evolution

This experiment began with a prompt to list and categorize status effects, followed by the construction of a tabular schema. The user then directed the model through a standard progression: JSON schema generation, Pydantic class implementation, serialization methods, and unit test creation. Each stage was completed in a single prompt without restructuring, aside from final code cleanup. Documentation was omitted. The task followed a typical flat-schema pipeline seen in other low-complexity cases and generalized well to similar domains.

Failure Modes and Debugging

No semantic or structural bugs were introduced by the model. The only required fix was to adjust import statements in the test file to match the module layout. No corrections were needed for class attributes, logic, or serialization behavior. This case shows that tightly scoped, data-driven prompts can produce valid and complete code with minimal debugging.

Task Complexity

This was a low-complexity task. The schema was flat, without nested structures or polymorphism. No game mechanics were implemented, only static attributes. The model performed well under these conditions, completing the task with minimal revision and high fidelity to the specification.