Vibe-Coding the Aristocrat: A Comparative Study of LLM-Assisted Cryptanalysis

Abstract

This report documents the development of a program designed to solve Aristocrat ciphers through an exploratory, conversation-driven methodology known as "vibe-coding." By engaging with two distinct Large Language Models (LLMs)—Google Gemini and OpenAI's GPT—as collaborative programming partners, I navigated the complex journey from initial brainstorming to a robust, automated cryptanalysis tool. This paper details the development narrative, provides a comparative analysis of the distinct architectural and philosophical approaches of the two AIs, and reflects on the critical lessons learned regarding human-AI partnership, software design, and the nature of modern problem-solving.

1. Approach and Tools

The goal of this project was to solve a monoalphabetic substitution cipher by applying cryptanalytic reasoning, not as a solitary developer, but in a dynamic partnership with modern LLMs. The "vibe-coding" approach was central to this experiment, prioritizing an open-ended, meandering dialogue over a rigid, pre-defined plan. This allowed the natural strengths and weaknesses of each AI collaborator to surface, providing rich material for comparison.

- LLM Collaborators: Google Gemini and OpenAI GPT.
- Programming Environment: Python 3, leveraging only standard libraries to ensure maximum portability and focus on algorithmic logic.
- Development Style: "Vibe-coding," which involved iterative dialogue, brainstorming sessions, collaborative implementation, and joint debugging with the LLMs.
- Primary Test Cipher:
  JBVZ DHK EX RZP, SCMQZ YZDZ GHVNM GRK GMGR HMQZ UDR DHK VZG HMQZ YPVZ XMZGRZ. RZP DHK SCMQZ KXGHZ UDR DHK KXGHZ LQZB GMGR KXGR UDR YPVZ RZP SCMGUR KSQMZ GHVNM. HMQZ YPVZ GMGR NMGUR ZXGHZ VZG XMQK SCMQZ DHK KSQMZ ZXGHZ GMGR SCMQZ DHK KXGHZ RZP.

2. The Interaction Narrative:

The development process was not a linear path but a parallel exploration of two distinct AI-driven methodologies. I presented the same initial problem to both Gemini and GPT and followed the divergent paths they proposed.

2.1. Session 1: Building a Clear Foundation with Google Gemini

My first session was with Gemini, where my goal was to establish a solid, understandable foundation. I started with a broad prompt: "Let's build a program to crack an Aristocrat cipher. How should we structure it?"

Gemini immediately proposed a clean, object-oriented architecture, suggesting the program's logic be separated into two distinct classes: a `LanguageModel` to handle statistical analysis and a `CipherSolver` to manage the search for the key. This approach was immediately appealing from a software construction perspective, as it was modular, extensible, and easy to reason about.

The code Gemini produced was heavily commented, explaining the purpose of each method and its design choices. A key feature was the decision to embed a training corpus directly into the script, meaning the language model was built from first principles every time. The core solving algorithm, a stochastic hill-climbing search, was implemented in a clear and easy-to-follow manner. This session felt like a true pair-programming partnership, culminating in a well-engineered and transparent tool.

2.2. Session 2: An Expert's Alternative with OpenAI GPT

To gain a comparative perspective, I posed the same problem to GPT. The result was a stark contrast in both architecture and philosophy. Instead of a modular, class-based structure, GPT generated a procedural script composed of numerous helper functions. Its approach to the language model was also different; rather than learning from a corpus, it utilized large, hard-coded dictionaries of pre-computed n-gram frequencies, prioritizing performance and compactness over transparency.

The solving algorithm was also more complex. GPT implemented a simulated annealing process—a more sophisticated search heuristic than simple hill-climbing—and added a final "steepest-ascent polish" step to refine the solution. While this code was functionally powerful and algorithmically dense, it was significantly less readable. The interaction felt less like a collaboration and more like a consultation with a domain expert who provides a highly optimized but opaque artifact.

2.3. The Final Decision: Prioritizing Software Construction Principles

After evaluating both generated programs, I made the conscious decision to adopt the Gemini-provided code as my final version. The choice was guided by the principles of this course: software construction. The Gemini code's modularity, clarity, and maintainability made it a superior example of well-designed software. The process of understanding, trusting, and potentially extending the code was far simpler. This foundation was then refined to produce the final `solver.py`deliverable.

3. Comparative Insights: Gemini vs. GPT

Google Gemini: The Software Engineer & Teacher

- Architectural Style: Favored a clean, Object-Oriented design that separated concerns into logical modules. This approach is highly valued in professional software development for its maintainability and scalability.

- Problem-Solving Philosophy: Its methodology was pedagogical. By building the language model from an explicit corpus, it created a system whose logic was transparent, effectively "teaching" how the solution worked from the ground up.
- Code Quality: Highly readable, well-commented, and robust. It prioritized clarity over cryptic optimization, resulting in a well-engineered tool.

OpenAI GPT: The Expert Algorithmist & Consultant

- Architectural Style: Opted for a procedural, function-based script common in academic or research code where the focus is on rapidly implementing a complex algorithm.
- Problem-Solving Philosophy: Its approach was that of an expert providing a finished solution. It used pre-computed data and advanced heuristics to achieve high performance, but at the cost of transparency. The code was a "black box" that worked effectively but was harder to dissect.
- Code Quality: Functionally very powerful and algorithmically sophisticated. However, its lack of modularity and reliance on hardcoded "magic numbers" made it less maintainable from a software engineering perspective.

4. Reflections and Conclusion

I was most surprised by the stark difference in the architectural choices made by the two models. I had anticipated variations in syntax, but not two fundamentally different design philosophies. Gemini approached the problem as a software engineer, while GPT approached it as a computational linguist. This revealed that the "best" solution is highly dependent on one's priorities—be it readability, performance, or transparency.

The "vibe-coding" process demonstrated that the developer's role is shifting from a pure implementer to that of a director, designer, and critic. My most important contribution was not writing lines of code, but asking the right questions, evaluating the proposed architectures, and making a critical design decision based on the principles of good software construction. The quality of the final product was a direct result of guiding the AI partners toward a desired outcome.

Finally, this experiment highlights a critical challenge in an AI-assisted world: reproducibility. While my final code is deterministic, the *process* of its creation is not. Another developer interacting with the same models might receive entirely different solutions. This makes documenting the interaction narrative—the "why" behind the code, including the paths not taken—an essential practice. Without this context, the creative and analytical process is lost, making true reproducibility impossible.

In conclusion, both Gemini and GPT proved to be exceptionally powerful collaborators, but their distinct styles necessitate a thoughtful and engaged human partner. The future of software construction will likely not be about which AI is "better," but about how developers learn to leverage the unique strengths of different models to achieve their goals, all while upholding the timeless principles of good design.