AI & SOCIETY
https://doi.org/10.1007/s00146-025-02786-w

RESEARCH

Leveraging LLMs for interpreting historical source code: a case study of the Apple Lisa through critical code studies

Titaÿna Kauffmann[1]

Received: 11 April 2025 / Accepted: 28 November 2025

Abstract
This study evaluates conversational large language models (LLMs) as pedagogical brainstorming tools for historical source code analysis through structured prompt-based approaches adapted from Critical Code Studies (CCS). The research tests whether conversational interfaces like ChatGPT-4o can support initial exploration of complex historical codebases by adapting CCS perspectives into conversational prompt formats. The dual-prompt evaluation separates technical parsing from interpretive reasoning, assessing how effectively conversational interfaces extract structural information while generating preliminary interpretive hypotheses. Using the Apple Lisa source code as a case study, this analysis documents both pedagogical utility and systematic limitations. The findings demonstrate that while conversational LLMs can preserve developer annotations, parse visual artifacts such as ASCII diagrams, and generate educationally valuable insights for approaching unfamiliar programming languages, the conversational user interface significantly constrains systematic analytical capabilities. Through three case studies examining ASCII typography, architectural diagrams, and interface implementations, the analysis illustrates how conversational interfaces function as structured brainstorming partners while necessitating rigorous validation against primary sources. This evaluation contributes to understanding AI's pedagogical role in digital humanities by positioning conversational LLMs as question-framing tools rather than interpretive authorities, while identifying systematic implementation requirements necessary for rigorous computational approaches to historical software analysis.

Keywords Large language models · Prompt engineering · Critical code studies · Apple Lisa · Digital history

1 Introduction

The concept of a Large Language Model (LLM) as a "Code Studies Advisor" was discussed during the *AI and Critical Code Studies* (CCS) week at the 8th biennial Critical Code Studies Working Group in February 2024 (CCS Working Group 2024). These discussions opened a provocative line of inquiry: could LLMs, typically associated with code generation or debugging, be adapted as analytical instruments to assist critical code reading—not merely interpreting what code does, but exploring how it means? This encounter revealed opportunities for LLMs to enhance research approaches that treat source code as historical text, examining it both as technical artifact and as expressive, culturally embedded document. Beyond research applications, this methodology offers significant pedagogical value as a brainstorming tool, providing structured frameworks for approaching unfamiliar codebases and technical languages.

This approach demonstrates that conversational LLM interfaces like ChatGPT-4o can be valuable teaching tools for initial exploration of historical code archives, while also revealing important limitations that require more structured approaches. Based on Daston and Galison's concept of epistemic virtues (2010), we tested various prompt strategies to use LLMs as structured research tools. However, the analysis identified significant problems with conversational interfaces, including factual hallucinations and inconsistent analytical frameworks. Although we aimed to assess whether LLMs could serve as reliable research tool similar to statistical software or text analysis programs, ChatGPT-4o's conversational limitations prevented us from implementing systematic analytical methods. This suggests that properly evaluating LLMs' potential as research tools would require API-based agent architectures rather than simple dialog interactions.

✉ Titaÿna Kauffmann
titayna.kauffmann@uni.lu

[1] Luxembourg Centre for Contemporary and Digital History (C²DH), University of Luxembourg, Esch-sur-Alezette, Luxembourg

Published online: 13 December 2025

Building on Gonzalez Garcia and Weilbach's (2023) "conversational methodology", this proof-of-concept approach demonstrates both the potential and limitations of employing LLMs as research brainstorming tools. While these models can facilitate dynamic inquiry processes and generate contextually informed responses when provided with specialized academic sources, the case studies reveal instances where the LLM produced plausible but unsubstantiated interpretations—particularly when operating on incomplete source material. This systematic evaluation provides historians with insights into how such tools might augment scholarly investigation while highlighting the necessity of rigorous verification protocols.

This methodology proves particularly valuable for examining the Apple Lisa system, released in 1983—a pioneering computer that introduced the first commercially available graphical user interface (GUI). The Lisa's historical significance extends beyond its technical innovations to encompass fundamental shifts in human–computer interaction paradigms. As Perkins, Keller, and Ludolph document, the Lisa development team created not merely a new interface but an entirely new conceptual framework for personal computing, establishing design principles and interaction metaphors that would define desktop computing for decades (1997). The Lisa represented a critical transition moment where computing moved from command-line abstractions toward visual, document-centered environments, making its source code a particularly rich site for understanding how software embodies cultural and technological transformation.

The Lisa Office System codebase—published in 2023 by the Computer History Museum—contains over a thousand files, comprising 614 Pascal sources, 203 assembly files, and various supporting materials (Computer History Museum 2023; Keller 2023). Written primarily in Pascal Lisa, a proprietary dialect developed specifically for the system's segmented memory model, this codebase presents both opportunities and challenges for contemporary researchers. This article focuses on the LisaDesk, a foundational component that implemented the Lisa's desktop metaphor and established the visual language that would define the graphical user interface (GUI). The estimate for the program reaches 275,000 to 290,000 lines, with developer annotations comprising nearly 40% of the total. These comments preserve traces of design decisions and implementation debates, transforming the codebase from mere instruction sets into sites of cultural inscription.

This paper presents a dual-prompt architecture for conducting systematic analysis of historically significant yet technically complex source code through conversational interfaces, while documenting the analytical limitations that emerge from this approach. We detail the methodology through three case studies from the LisaDesk, examining ASCII typography, architectural diagrams, and interface implementations. These examples demonstrate both the pedagogical utility of LLM-assisted brainstorming and the critical need for verification protocols, revealing how this approach can illuminate technical and cultural dimensions of historical software while raising important questions about AI as a supportive tool in research. The cases highlight the method's particular value for the initial exploration of unfamiliar programming contexts while underscoring the systematic implementation requirements necessary for rigorous scholarly application.

2 Leveraging AI as a research tool

LLMs have evolved from theoretical constructs into potentially useful analytical support systems for scholarly research. While researchers have primarily used LLMs for summarization and text generation, recent studies are exploring their potential in historical analysis (Oberbichler and Petz 2025; Karjus 2024). This approach evaluates emerging work in digital humanities that examines whether AI conversational interfaces can function as pedagogical brainstorming tools that support rather than substitute human interpretive expertise. Through testing specialized prompts within ChatGPT-4o's conversational framework, we assess the interface's pattern-recognition capabilities while documenting the verification challenges and analytical limitations that emerge from this approach.

This exploratory study combines conversational LLM interaction with academic sources to evaluate structured analytical processes within interface constraints. Garcia and Weilbach (2023) demonstrate that scholar–machine interaction can serve as a research accelerator, though the findings reveal significant limitations when attempted through conversational interfaces rather than systematic implementation. This methodology tests recent advances in prompt engineering, specifically *Chain-of-Thought* and *Tree-of-Thought* architectures, adapting them for two complementary analytical modes: technical parsing and critical interpretation. This dual-prompt evaluation addresses practical constraints of conversational interfaces while assessing whether such approaches can preserve code's evidentiary specificity and enable meaningful interpretive analysis of both technical architecture and embedded socio-cultural dimensions.

The first prompt functions as a technical analysis tool calibrated to parse Lisa code through the computing environment of the early 1980s. This analytical configuration attempts to decode functional logic, identify architectural patterns, and highlight significant features that would remain opaque to non-specialist readers. The second prompt reconfigures the framework to explore critical interpretation possibilities, generating potential avenues for historical theorization and cultural analysis that researchers might systematically investigate. Rather than producing definitive scholarly conclusions, this dual-prompt evaluation provides structured outputs that reveal both the pedagogical utility and verification requirements of conversational LLM interfaces.

However, this evaluation reveals that conversational interfaces present significant challenges for systematic analytical work. As Berry (2023) emphasizes, the integration of LLMs into digital humanities research requires maintaining critical scholarly distance and methodological transparency. Moreover, as Messeri and Crockett (2024) observe, LLMs can create “illusions of understanding”—instances where persuasively written outputs conceal analytical gaps or fabricated content. This phenomenon, commonly referred to as hallucination, occurs when models generate plausible-sounding interpretations without factual foundation. According to Huang et al. (2025), hallucinations emerge from misalignment between a model’s internal processes and semantic accuracy.

Drawing inspiration from Lévi-Strauss’s insight that “the scientist is not the one who gives the right answers, but the one who asks the right questions” (1964), we evaluate whether conversational LLMs can function as question-framing tools rather than answer’s providers. In the context of source code analysis, the assessment examines whether such interfaces can support historians’ interpretive exploration—particularly when encountering artifacts that are simultaneously technically complex and culturally significant, while documenting the analytical constraints that emerge.

The evaluation methodology encompasses both interface assessment and operational protocols to test conversational LLM deployment in historical research. At the technical level, we employ structured prompts within ChatGPT-4o’s conversational constraints. At the operational level, we establish verification protocols where the historian functions as output evaluator and interpretive authority. This evaluation operates through three mechanisms: first, testing analytical separation through distinct conversational prompts; second, assessing iterative processing capabilities within interface limitations; and third, documenting verification requirements where all outputs require cross-referencing against primary sources. By treating conversational LLM responses as preliminary brainstorming material rather than analytical conclusions, we assess both pedagogical utility and systematic limitations.

We selected ChatGPT-4o for evaluation due to its accessibility and consistent conversational performance, though the findings emphasize that conversational interfaces present inherent constraints for systematic research application. While prompt engineering offers transferable methodological insights across various LLMs, the evaluation reveals that ChatGPT-4o’s conversational framework, like other foundation models, functions as a black-box system with opaque reasoning processes that necessitate extensive verification protocols. The following section details the prompting methodology, its adaptation from existing CCS literature, and the analytical challenges that emerged during conversational interface testing.

3 Parsing the Lisa’s structure

3.1 Historical context

The Apple Lisa emerged from a deliberate attempt to transform personal computing from a specialist tool into an accessible instrument for typical office workers. Formed in 1979, Apple’s Lisa development team confronted a fundamental limitation of contemporary systems: while they possessed adequate functionality, they lacked the capacity, speed, and intuitive operation necessary to serve users unwilling to master complex technical procedures (Birss 1984). This challenge positioned the Lisa project at a critical juncture where the industry needed to transition from command-line interfaces toward more natural interaction paradigms.

Drawing inspiration from Xerox’s SMALLTALK environment, the Lisa team developed an integrated system prioritizing direct object manipulation over abstract command structures. The interface philosophy centered on real-world metaphors rather than computer concepts, with a desktop that literally “mirrors the function of a desk in the office” through icons representing familiar objects like documents, folders, and wastebaskets (Birss 1984). As the development team documented, users could manipulate objects directly through mouse interaction rather than memorizing textual commands (Perkins et al. 1997). Apple positioned this as fostering “a more productive, efficient office environment by following the work habits of the individual. The result is a system that functions as a natural extension of its user” (Apple Computer Inc. 1983). This represented a fundamental shift from viewing computers as calculating machines toward conceptualizing them as extensions of existing work environments, supported by multitasking capabilities that enabled fluid movement between applications.

The system was programmed primarily in *Pascal Lisa*, a custom dialect that preserved the procedural clarity of *UCSD Pascal* while adapting to Lisa’s unique hardware requirements. Apple’s modifications accommodated the segmented memory architecture through unit-based modular compilation, fine-grained segment control, and deep integration with system libraries handling graphics, alerts, file systems, and memory management.

To address the interpretive challenges posed by this legacy code with its proprietary language modifications, this research tested whether conversational LLMs could serve as useful support tools for initial code exploration and interpretation. We evaluated their capacity to process both formal syntax and natural-language commentary within conversational interface constraints, assessing whether such approaches could facilitate identification of technical structures, stylistic patterns, and embedded design decisions while navigating obsolete development environments. To explore this pedagogical potential, we constructed a specialized technical software analyst prompt configuration for ChatGPT-4o, though the findings reveal significant limitations in systematic analysis capabilities that would require more rigorous implementation frameworks to address.

3.2 Technical analysis configuration

To evaluate the Lisa codebase's accessibility through conversational LLM interfaces, we tested structured prompting strategies that attempted to establish analytical parameters, output specifications, and processing methodology within ChatGPT-4o's conversational constraints. This experimental configuration employed explicit role priming and contextual grounding to assess whether conversational interfaces could support historical Pascal and assembly code interpretation, while documenting the verification challenges that emerged.

The configuration began with articulated persona assignment—a documented technique for attempting to align LLM behavior with domain-specific analytical requirements (Garcia and Weilbach 2023; Yin et al. 2024):

```
The AI is an expert in Technical Software Analysis, analyzing historical source code
for functionality, structure, and design principles.
The AI will analyze the Apple Lisa source code (1983) to extract its technical
structure, functions, and developer comments, ensuring all data is captured.
```

This explicit role attempted to establish an analytical framework while anchoring the model's responses within the specific temporal and technical context of the 1983 Apple Lisa ecosystem—its Pascal and 68,000 assembly codebase, GUI applications like LisaWrite and LisaCalc, and hardware constraints. While this contextual framing draws on research suggesting domain-specific prompting can improve outputs (Besta et al. 2024; Karjus 2024; White et al. 2025), the evaluation revealed that conversational interfaces struggled to maintain consistent analytical frameworks, producing interpretations that required extensive verification against primary sources.

Given the critical importance of preserving developer annotations as primary historical evidence, we established explicit fidelity constraints:

```
 Preserve all developer comments exactly as they appear, including formatting,
indentation, and punctuation.
- Never summarize, truncate, or paraphrase developer comments.
- Do not use ellipses (`...`). Always reproduce the full original block.
- Retain misspellings and spacing — treat comments as primary historical data.
```

However, the testing revealed that conversational interfaces struggled to maintain these constraints, particularly when processing longer code segments that approached the interface's context window limitations, necessitating continuous cross-verification against original sources to ensure historical accuracy.

We framed the core analytical task as technical translation rather than simplification:

```
Translate Pascal code into structured, technically accurate English.
Identify and explain functions, procedures, constants, types, and comments.
Preserve the code's technical and historical intent without simplification.
```

While these specifications aimed to preserve informal commentary, from terse system instructions to expressive implementation notes, as historical evidence, the evaluation demonstrated that conversational interfaces could generate useful initial interpretations while requiring systematic verification protocols. The approach proved particularly valuable for parsing unfamiliar proprietary syntax variations and providing accessible entry points into complex historical codebases.

The configuration successfully parsed visual structures within the code (ASCII diagrams, memory maps, and flowcharts) which often serve as internal documentation of the system's conceptual model. This capability addressed documented LLM limitations with preserving structural and contextual nuance (Garcia and Weilbach 2023; Wagner et al. 2025; Karjus 2024), producing meaningful technical translations that facilitated initial code comprehension, though systematic verification remained essential for scholarly application.

The evaluation employed *Chain-of-Thought with Self-Consistency* methodology, testing whether conversational interfaces could support structured reasoning sequences (Wei et al. 2022). The model demonstrated capable systematic identification of functions, procedures, types, and constants through detailed breakdown protocols, generating technically coherent analyses that provided valuable pedagogical insights, though certain analytical inconsistencies emerged that we address in subsequent sections.

This configuration successfully tested technical translation capabilities within conversational constraints, rendering Pascal Lisa's proprietary syntax into accessible structured English while preserving logical clarity. The approach demonstrated significant pedagogical utility for initial code exploration and proved effective for generating preliminary technical documentation.

4 Developing an interpretive framework

Following technical parsing, we reconfigured ChatGPT-4o to operate as a CCS analytical support tool, testing whether conversational interfaces could effectively examine the Lisa codebase as both technical artifact and cultural–historical document. Following the CCS tradition as outlined by Marino (2020), this second analytical phase processes the structured technical output to explore source code as a discursive object containing cultural logics, rhetorical strategies, and historical traces. Building on the first-phase findings of function definitions, data structures, and developer comments, this configuration successfully shifted the interface's analytical focus from technical functionality to interpretive significance, demonstrating how conversational LLMs can support exploration of how code shapes interaction, reflects assumptions, and embeds ideological frameworks.

The reconfiguration establishes a specialized interpretive framework calibrated for humanities-oriented code analysis:

```
The AI is an expert in Critical Code Studies (CCS), analysing software as both a
technical artifact and a cultural-historical document. To do so, they will use the
results from the expert in Technical Software Analysis.
```

Operationally, the model processes the first prompt's structured output as curated input for deeper interpretive analysis. This approach addresses concerns scholars have raised about AI's tendency to strip meaning from historical context (Bender et al. 2021; Messeri and Crockett 2024) by grounding analysis in the system's original temporal and technological setting, building upon the historical background established earlier in Appendix A's description of the Lisa ecosystem.

The framework draws upon a genealogy of interdisciplinary work that has progressively framed code as both technical and cultural artifacts. This tradition spans Manovich's *The Language of New Media* (2001), which introduced performative analysis of user interaction and experience; Mackenzie's *Cutting Code* (2006), which contributed sociological insights about organizational practices and labor structures; and Fuller's *Software Studies: A Lexicon* (2008), which developed materialist perspectives on software's infrastructural conditions. These approaches converge in Mark C. Marino's 2006 manifesto for Critical Code Studies, which called for integrating hermeneutic methods into code analysis as historically and culturally embedded text.

As described below, we formalized these theoretical perspectives into three complementary interpretive lenses for LLM comprehension: performative analysis examines how code structures user interaction and distributes agency; cultural analysis explores how code embeds institutional values, naming conventions, and developer worldviews; and materialist analysis investigates the infrastructural, ecological, and labor-intensive dimensions of software systems. These categories function as flexible analytical heuristics rather than rigid taxonomies, with author citations stripped from the prompt structure to allow direct engagement with the code's underlying logic.

To structure this interpretive process, we adopted the *Tree-of-Thought* reasoning, a prompting technique that elicits reflective and branching chains of reasoning from large language models. As Besta et al. (2024) explain, this framework transcends linear answer generation by encouraging models to explore multiple solution paths simultaneously, constructing tree-like structures of hypotheses, evaluations, and revisions. This approach proves valuable for humanities-oriented tasks where ambiguity and multiplicity are essential features rather than flaws to be eliminated.

Aligning with CCS's emphasis on interpretive openness (Marino 2020), the implementation directs the LMM to generate multiple interpretive branches per code element, each offering distinct analytical perspectives:

```
Before interpretation, assess which of the three perspectives—performative, cultural,
or materialist—best applies to the selected code element.

Select one or two perspectives that offer the strongest explanatory value. Exclude
perspectives that do not meaningfully apply. Briefly justify your selection.
```

Critical to maintaining analytical rigor, the configuration includes systematic uncertainty handling protocols:

```
When faced with ambiguity in comments or behavior:

1. Acknowledge the uncertainty.

2. Offer at least two competing hypotheses.

3. Rank them based on plausibility.

4. Justify your ranking using evidence from both the structured analysis and the
original source code.

5. Note what additional historical or technical sources would help resolve the
ambiguity.
```

This approach mirrors interpretive reasoning in humanities scholarship, where multiple meanings and indeterminacy constitute analytical strengths. The LLM operates by processing the technical analysis as curated input while consulting original source code for verification and detail recovery. Rather than functioning as an interpretive authority, this configuration generates tentative, evidence-based readings that require subsequent scholarly validation through cross-referencing with primary sources, contemporary documentation, and established CCS scholarship—standard practices in historical research methodology.

4.1 The performative dimensions of code

The performative analysis component configures the LLM to examine code through Manovich's framework (2001), treating software as both computational process and cultural mediator. This analytical mode operates through dual interpretive layers: the computational level, where code enacts processes by structuring data, logic, and interface interactions; and the cultural level, where these processes function as mediational gestures that configure user engagement patterns. These layers operate through what Manovich terms "cultural transcoding"—a recursive dynamic where technical constraints shape cultural expression and vice versa.

The analytical framework proved particularly effective for examining the Lisa's early GUI design decisions, where code both enabled and constrained emerging user interaction paradigms. The interface's technical parsing capabilities reveal embedded assumptions about user behavior and computational agency, demonstrating how source code functions as an active participant in shaping interface conventions and software paradigms rather than merely executing computational routines.

The performative analysis configuration employs the following three-step interpretive process:

```
Steps:
1. Identify code elements (e.g., functions, UI handlers, control structures) that
govern user interaction or system behavior.
2. Explore how the code embodies cultural practices or redefines user roles.
3. Connect these behaviors to broader computing trends and compare them to
contemporary systems (e.g., Xerox Star, IBM PC).
```

This methodology treats computational routines as interface gestures encoded in logic. The prompt analyzes how code elements configure or reconfigure user agency, examining cultural assumptions embedded in design decisions—considerations of ease-of-use, immediacy, and spatial navigation. Finally, it contextualizes these behaviors historically through comparisons with contemporary computing systems.

To calibrate the analytical framework, the configuration includes paradigmatic examples for performative reasoning:

```
Example:
- Direct Code Examination: { if userInput = "exit" then CloseAllWindows(); }
- Interpretations:
  - Path 1: Reinforces GUI as a spatial computing metaphor.
  - Path 2: Embodies Apple's intuitive design philosophy.
- Conclusion: Demonstrates real-time, user-driven control, aligning with early GUI
principles.
```

The prompt utilizes *Tree-of-Thought* branching (Wei et al. 2022; Besta et al. 2024) to generate multiple interpretive paths for code fragments. The example routine—though computationally simple—helps encapsulate spatial metaphors and user-centric design principles characteristic of early GUI paradigms. The analytical framework treats such routines as performative gestures that reinforce desktop metaphors, generating interpretive branches that examine spatial coherence requirements and interface closure rituals aligned with intuitive design philosophies.

4.2 The cultural analysis of code

The cultural analysis component shifts analytical focus from what code *does* to *how* code operates within social, political, and economic systems. This configuration treats software not as neutral technical instrument but as a medium that distributes agency, reinforces institutional priorities, and encodes values through design, implementation, and circulation. Drawing from Adrian Mackenzie's *Cutting Code* (2006), this analytical framework positions software as social practice shaped by organizational conflicts, norms, and institutional imaginaries rather than merely logic formalized in syntax.

The cultural analysis configuration instructs the LLM to trace how developer decisions, naming conventions, comments, and system constraints reflect organizational culture, policy concerns, and broader power dynamics:

```
Steps:
1. Examine how the code embeds cultural values.
2. Explore how the code encodes corporate priorities, labor practices, and societal
influences.
3. Connect these insights to historical contexts by analyzing their relationship to
workplace culture, governance models, and contemporary debates on labor, technology,
gender roles, and accessibility.
```

This methodology guides the LMM to move beyond functional description toward examining embedded institutional values. The configuration explores how technical features encode corporate priorities, labor practices, or assumptions about user roles, and then situates these elements historically within 1980s workplace computing trends.

To calibrate the analytical framework, the configuration includes paradigmatic examples for cultural reasoning:

```
Example:
- Direct Code Examination: { **** Modification Log **** 12/20/83: Added password
checking - Steve }
- Interpretations:
  - Path 1: Reflects security policies as computing shifted to enterprise markets.
  - Path 2: Suggests increasing control over user access.
- Conclusion: Highlights tensions between security and openness in early computing.
```

The prompt aims to guide the model in examining seemingly mundane annotations as cultural artifacts. Given that the global objective is to uncover potential avenues for the cultural approach that will need to be cross-referenced later, the main goal is to reveal how technical annotations may reflect growing corporate concerns with user access control.

By directing the LMM to produce multiple historically contextualized interpretations, this configuration emphasizes the inherently interpretive nature of cultural code analysis. Given that the evaluation aimed to uncover potential avenues for cultural analysis that could provide productive starting points for subsequent scholarly investigation, the approach proved effective at revealing how technical annotations might reflect broader corporate concerns with user access control and institutional priorities.

4.3 The materiality of code

The materialist analysis component reframes code as situated, operational practice deeply embedded within technological, economic, and ecological infrastructures rather than pure logic or symbolic abstraction. Drawing from Matthew Fuller's perspective (2008), this configuration treats code as actively reorganizing and responding to computation's physical limits—regulating memory, scheduling processes, coordinating I/O operations, and shaping human–machine interaction rhythms. Code exists within its material context as what Fuller terms "operative writing" that co-produces the systems it inhabits.

The materialist analysis configuration directs the model to isolate technical structures that interface directly with hardware constraints or embody resource optimization strategies:

```
Steps:
1. Examine how the code manages system calls, memory routines, and resource
allocation.
2. Explore how the code encodes efficiency strategies, labor practices, and
computational constraints.
3. Connect these insights to historical contexts by analyzing their relationship to
1980s hardware constraints, Apple's software design, and coding labor.
```

This methodology operates through a three-phase analytical process: identifying relevant low-level mechanisms; generating interpretive paths regarding their labor and infrastructural implications (Besta et al. 2024); and contextualizing insights within early 1980s computing conditions. This approach could provide rich analytical terrain given the Lisa's operation on a Motorola 68,000 processor managing complex GUI interactions under strict memory and performance constraints.

To calibrate this analytical framework, the configuration includes paradigmatic examples for materialist reasoning:

```
Example:
- Direct Code Examination: { ADD.W D1, MemoryAllocationTable }
- Interpretations:
  - Path 1: Demonstrates low-level efficiency strategies for resource-limited
environments.
  - Path 2: Embodies the labor-intensive nature of early coding.
- Conclusion: Reflects the entanglement of software with hardware constraints and
material computing practices.
```

This technique guides the model to examine seemingly technical instructions as vectors of infrastructural tension, indexing both machine performance requirements and human labor invested in extracting functionality from physical limitations. The analytical framework considers how routines are shaped by—and shape—coding labor practices, resonating with Mackenzie and Vurdubakis's (2011) argument that code performs in excess of itself, entangling semiotic, economic, and bodily domains.

In the Lisa context, such entanglements include optimization labor, invisibilized display management infrastructures, and symbolic density of procedures animating the graphical desktop. This positions the analytical prompt to examine software as historically embedded artifact of "computation under constraint," providing systematic analysis of how code functioned within material limitations and what computational world it helped construct, requiring subsequent validation through hardware documentation and contemporary programming practice records.

5 Evaluating the results: three applied case of the prompt

This section showcases how LLMs can function as exploratory pedagogical instruments for identifying previously overlooked elements. Rather than systematic evaluation, we present three demonstrations of the structured prompting framework, including LLM outputs alongside human interpretation to illustrate how these tools can support historians in brainstorming preliminary research directions or serve as pedagogical tool in academic settings. Following Wagner et al. (2025)'s emphasis on methodological transparency in LLM-assisted research, complete conversation logs and prompt sequences are publicly available in the research repository.[1]

The LisaDesk software is part of a significant moment in the historical transition from command-line to graphical user interfaces, encoding the translation of abstract file operations into spatial metaphors within the technical constraints of early 1980s personal computing. This software component, documented by the development team as central to implementing the desktop paradigm, provides a focused case study for demonstrating how LLM-assisted analysis can identify previously unexamined textual elements within historical source code.

The three case studies below illustrate different CCS approaches—performative, cultural, and materialist. Each case demonstrates how these pattern-recognition tool can

[1] Complete LLM conversation logs and prompt documentation are available at: https://github.com/titayna/apple-lisa-ccs-analysis.

generate preliminary insights that require scholarly validation and theoretical grounding.

5.1 The art of code: ASCII typography in Lisa's desktop manager

The dual-prompt methodology generated systematic analytical outputs for examining ASCII typography within the Lisa Desktop source code, more precisely the APDM-DESK.TEXT.unix file. The technical analysis prompt identified a stylized ASCII art block embedded at the program's beginning, capturing visual structures as integral codebase components rather than peripheral marginalia:

The LLM's analytical framework correctly identified cultural analysis as the most applicable interpretive lens while appropriately excluding performative and materialist approaches that lacked sufficient contextual anchoring for this particular code element. The prompt generated two coherent interpretive branches that address different aspects of ASCII art's function within early software development culture.

However, the output requires historical contextualization and validation through established scholarship. The LLM's interpretation of ASCII art as "corporate branding" and "authorial marking" aligns with the existing literature on textual performance within software culture. As

```
Lines 3–16: Developer Comment Block
   {
            L                        DDDD
            L      o                 D   D                 k
            L         sssss  aaaaaa  D    D  eeeee  sssss  k  k
            L      i  s      a    a  D    D  e      s      k k
            L      i  sssss  aaaaaa  D    D  eeeee  sssss  kk
            L      i      s  a    a  D   D   e          s  k k
            LLLLLL i  sssss  a    a  DDDD    eeeee  sssss  k  k
                                Copyright 1983, 1984, Apple Computer Inc.

   }

Commentary:
- This is an ASCII art title card stating "Lisa Desktop", followed by the copyright.
- The art serves both as branding and a visual delimiter.
- Maintains historical authenticity.
```

Building upon this technical foundation, the critical code studies analysis prompt generated is the following:

```
Selected Perspective(s): Cultural
Excluded: Performative, Materialist
Interpretive Branches:
- 1. ASCII Art as Corporate Branding
The Lisa Desktop ASCII banner performs a dual function: visually branding the software
and signaling internal identity among developers. This playful insertion into a
typically austere medium (Pascal source code) reflects early 1980s Silicon Valley
ethos—merging countercultural aesthetics with corporate coding practices.
- 2. Authorial Marking and Labor Visibility
By embedding an ASCII title card, the developers mark authorship in a domain where
code was often anonymous. This counters the invisibilization of software labor
especially notable during an era when Apple was scaling rapidly and consolidating
control over its intellectual property.
Synthesis:
This element reflects the hybrid culture of early Apple: blending hacker-style
expressivity with burgeoning corporate formalism. It embodies a liminal space where
countercultural roots met the emerging discipline of professional software
engineering.
```

Wardrip-Fruin and Montfort (2003) demonstrate, ASCII art represents a unique intersection of creative expression, technical documentation, and developer culture, where code becomes a site of aesthetic inscription beyond functional logic. The interpretive framework identifying this as "textual performance" corresponds with Cox et al.'s analysis in *The Aesthetics of Generative Code* (2001), which positions code as performance with intrinsic aesthetic dimensions.

The LLM's second interpretive branch concerning "labor visibility" requires more cautious evaluation. While the analysis correctly identifies ASCII art as asserting programmer presence, the broader claims about Apple's intellectual property consolidation and labor invisibilization demand verification through contemporary corporate documentation and developer accounts. The interpretation draws reasonable connections between aesthetic practices and workplace culture, but such claims exceed what can be definitively concluded from the ASCII art alone.

Cayley's (2004) concept of "literal inscription" provides additional analytical framework for understanding how this typographic intervention challenges computational transparency ideals. Even through the Lisa's whole design philosophy is aimed at concealing underlying code from users, ASCII banner, however, does the opposite: it deliberately asserts code's visibility, highlighting its material and typographic presence on the textual surface.

The model also generated an extended analytical output, demonstrating its capacity for deeper interpretive analysis:

```
1. Developer Comment Block (Lines 3–16)
Cultural Interpretive Branches (Expanded)
• 1. Corporate Graffiti
The ASCII art sits at the intersection of formal copyright claim and informal
authorship—corporate branding meets hacker subculture. It suggests a liminal space in
Apple's identity: between DIY garage ethos and industrial-scale professionalism. Code,
in this case, becomes a canvas for affiliation and pride—marking territory as much as
functionality.
• 2. Code as Aesthetic Medium
Choosing to adorn the code with stylized typography shows that developers weren't just
writing software—they were creating a cultural artifact. This kind of aesthetic
insertion hints at an early recognition of code's rhetorical power—as something read,
interpreted, and identified with.
• 3. Labor Recognition in a Post-Fordist Tech Economy
While 1980s software development was increasingly modular and anonymized, this comment
performs resistance to invisible labor. It marks authorship within a bureaucratic
product. This parallels broader shifts in post-Fordist economies where workers sought
symbolic ownership in increasingly fragmented, corporate systems
```

While the LLM's initial analysis remained grounded in observable textual features, this extended output reveals a tendency toward theoretical elaboration that exceeds what the ASCII art evidence can reasonably support. The introduction of "post-Fordist" frameworks and claims about "resistance to invisible labor" demonstrate how sophisticated analytical capabilities can generate plausible but historically unsubstantiated interpretations when the instrument attempts more complex reasoning. This highlights a key limitation: LLMs may produce increasingly elaborate theoretical constructs when prompted for deeper analysis, potentially moving further from empirical grounding rather than closer to it.

The analytical outputs demonstrate the LLM's capacity to generate structured interpretive possibilities while highlighting the necessity of subsequent scholarly validation. The cultural analysis produced meaningful insights into symbolic and rhetorical functions, though claims about broader institutional practices require cross-referencing with historical documentation of early Apple development culture and 1980s software engineering practices.

5.2 Visual logic: mapping the Lisa's desktop architecture in ASCII

The dual-prompt methodology encountered a more complex analytical challenge when examining an elaborate ASCII diagram embedded within the same file APDM-DESK.TEXT.unix. The technical analysis prompt successfully identified and parsed this visual representation:

Lines 316-353: ASCII Art — Object Hierarchy

```
      {
          deskObject
              !
              v
            -----
            !   !
            -----         Objects on desktop
             ! ^
             ! !<------------------------------
             ! !        ^                    ^
             v !        !                    !
            -----     -----                -----
        --> !h/t! --> !   ! -->         -->! f ! --> to h/t
        <-- !   ! <-- !   ! <--   ...   <--!   ! <-- from h/t
            -----     -----                -----
              !         !                   ! ^              contents of "f"
              v         v                   ! !
           nilObject                        ! !<-----------------------------
                                            ! !        ^         ^         ^
                                            v !        !         !         !
                                           -----     -----     -----     -----
                                       --> !h/t! --> !   ! --> !   ! --> !   ! -->
                                       <-- !   ! <-- !   ! <-- !   ! <-- !   ! <--
                                           -----     -----     -----     -----
                                             !         !         !         !
                                             v         v         v         v
                                          nilObject
     }
```

Diagram Interpretation:

- Depicts desktop object container model:
  - deskObject is the root.
  - Uses head/tail nodes (h/t) for circular doubly-linked lists.
  - Each folder (f) has its own linked object structure.
  - Arrows and labels describe: • Navigation and relationship pointers: next, prev, contents.
  - Recursive folder structure and traversal logic.
  - nilObject indicates end of lists.

This technical parsing successfully captured the diagram's architectural logic, identifying it as a circular doubly linked list structure that organizes desktop objects in memory. The prompt correctly recognized the hierarchical relationship between desktop objects and folder contents, demonstrating its capacity to interpret complex visual documentation as integral technical data.

However, the critical code studies analysis initially failed to generate interpretive output for this diagram, requiring methodological intervention before producing it. We had to nudge the model to apply the CCS lens to the specific lines identified in the first parsing, which produced the following result:

```
Diagrammatic Comment — Lisa's Desktop Object Model

Path 1: Diagram as Care and Communication

This diagram isn't just technical — it is pedagogical and artistic:

   • The coder is clearly imagining a future reader who will inherit this code.

   • The arrows and flow paths simulate pointer logic but do so spatially — making it
possible to visualize the circular, doubly-linked lists and container hierarchies.

   • Labels like "deskObject", "h/t", and "nilObject" translate memory pointers into
narrative actors.

This is software as storytelling, where nodes and pointers become characters in a
dynamic scene.

Path 2: Performative Blueprint for the GUI

This diagram is also the performative skeleton of the GUI:

   • It lays bare how every visible item on the Lisa desktop is represented internally.

   • The "deskObject" is a root node that performs the role of a visual stage manager.

   • Circular doubly-linked lists (h/t nodes) manage icons in each container, echoing
scene graphs used in modern UI frameworks.

Path 3: Bridging Embodied and Abstract Worlds

This ASCII diagram is a liminal artifact: it sits between language and image, between
machine and mind. It's not executable code, nor is it user-facing interface. It exists
in the in-between — the symbolic infrastructure of the program.
```

The initial analytical failure reveals important methodological limitations: when confronted with non-executable visual documentation, the cultural analysis framework struggled to apply its standard interpretive categories. This indicates that LLM-based analytical tools require more systematic calibration to properly handle all representational forms found in source code.

The subsequent analysis, while generating compelling interpretive frameworks, demonstrates both strengths and weaknesses in LLM-assisted interpretation. The characterization of the diagram as "software as storytelling" where "nodes and pointers become characters in a dynamic scene" offers insight into how visual documentation functions pedagogically within developer culture. This aligns with established scholarship on code as communicative practice, particularly Cox and McLean's analysis in *Speaking Code* (2012), which positions code as simultaneously operating as logic and aesthetic expression.

However, the analysis overstates its claims about historical significance, describing this as representing a "lost genre" in software development without sufficient evidence that such practices were widespread or subsequently abandoned. The interpretation draws reasonable connections between the diagram's visual logic and the Lisa's broader design philosophy of making computing "legible, humane, and metaphorically rich," but extends this into broader cultural claims that exceed what can be demonstrated from a single code artifact.

The materialist analysis proved less applicable, as the LLM correctly recognized that non-executable documentation lacks the resource constraints and optimization strategies central to that interpretive framework. This demonstrates appropriate analytical discrimination, though it highlights challenges in translating Fuller's concept of "boundary forms" from *Software Studies: A Lexicon* (2008) into systematic prompt-based analysis.

The diagram does function as what Herrenschmidt describes in *Les Trois Écritures* (2007) as combining instruction and expression, transforming abstract memory structures into visual form that invites readers to mentally navigate the system architecture. This pedagogical dimension reflects the Lisa development team's commitment to code readability and future maintainability, though broader claims about this representing unique cultural practices require validation through comparative analysis of contemporary software development documentation practices.

5.3 Methodological boundaries: when analysis exceeds available evidence

The dual-prompt approach revealed significant limitations when applying the materialist approach to the LisaDesk source code, more precisely the APDM-DESK2.TEXT.unix file. These limitations stemmed primarily from the framework being confined to a specific section of the code and its interconnections. Since the code does not exist independently of its hardware implementation, the LLM focused on these aspects as directed by the prompt. This partial access ultimately led the model to hallucinate part of its answer.

First, the technical analysis prompt successfully identified the UnfileObject function within the available code corpus:

```
Function: UnfileObject
- Animates object being dragged to the desktop (unfiled).
- Reassigns it to deskObject.
- Performs visual drag (WalkObject) and logical reassignment (MoveObject).
- Handles failures with fallback UI updates (ChangeObject, DrawObject).
```

The fundamental methodological challenge emerged from this identification: the prompt detected references to sophisticated animation and object management routines—WalkObject, MoveObject, ChangeObject, and DrawObject—which suggested significant materialist dimensions in graphics operations and resource management. However, these specific function implementations were absent from the provided source material.

This absence stems from the Lisa Office System's modular architecture, where the Desktop Manager functions as just one component within a larger framework. Lisa's source code was distributed across multiple Pascal units and compilation segments, with graphics routines, animation functions, and low-level drawing operations residing in separate source files aligned with different functional domains.

Building upon this partial technical foundation, the critical code studies analysis prompt attempted to apply materialist and performative frameworks to the UnfileObject function. The LMM's exposure to the broader LisaDesk codebase had revealed legitimate traces of hardware-constrained implementation—graphics port management, coordinate calculations, and memory handle operations—that would indeed support materialist analysis.

The result correctly identified the code excerpt cited by the technical analysis, but failed to render the function in its entirety, most likely due to the limited context window allowed by the user interface:

```
Direct Code Excerpt:
UnfileObject WalkObject(obj,0,0,dh,dv,deskPort); { animate object to desk
location }
...
IF NOT MoveObject(...) THEN
   ChangeObject(obj,normal,0,0,TRUE); { change back to normal }
[…]
```

More problematically, the materialist analysis proceeded to make specific claims about computational operations:

```
Materialist Analysis:
1. Locational Fidelity:

- The delta between homePt and deskPt is explicitly computed and animated.
- This reflects the tight coupling of visual location with logical state,
requiring system resources to uphold spatial metaphors.
```

However, in the result for the UnfileObject implementation, the LMM hallucinated access to code segments it had not actually received. Recent research by Kalai et al. (2025) identifies this phenomenon as characteristic of LLM behavior when operating on incomplete information: the model's pattern-recognition capabilities generate plausible-sounding completions based on learned associations rather than observable evidence, leading to what they term "confident fabrication" of specific details that align with broader contextual patterns.

The model's exposure to the LisaDesk codebase revealed legitimate evidence of coordinate calculations, animation parameters, and resource management strategies within the Desktop Manager orchestration logic. However, when analyzing UnfileObject specifically, the tool extrapolated from these patterns to make definitive claims about computational operations, it could not observe in the actual function implementations, which remained in separate, unavailable source modules.

The assertion that "delta between homePt and deskPt is explicitly computed and animated" exemplifies this methodological failure. While the prompt identified materialist dimensions within the available code—coordinate transformations, graphics operations, and memory management—it projected these patterns onto specific implementations without access to the actual computational details housed in separate libraries. The tool confused knowledge of general Lisa system patterns with empirical observation of specific algorithmic behavior implemented elsewhere.

This example illustrates what Huang et al. (2025) call a "factual hallucination" (making specific claims contradicting available evidence) and "reasoning hallucination" (logical inferences exceeding evidentiary support). The tool confused general knowledge about Lisa system architecture with direct empirical evidence of specific algorithmic functions implemented elsewhere in the system structure.

The tool's capacity to identify relevant theoretical frameworks based on partial code exposure proves valuable for preliminary analysis, yet this same pattern-recognition capability generates false confidence when examining specific implementations whose details remain distributed across unavailable source modules. Kalai et al. (2025) identify this as the "knowledge boundary problem," where models exhibit highest hallucination rates at the edges of their training distribution. The prompt detected legitimate traces of materialist-relevant operations throughout the available Desktop Manager source code, leading it to assume similar operations existed within supporting functions it could reference but not fully analyze.

6 Conclusion

Drawing from Lévi-Strauss's insight that "the scientist is not the one who gives the right answers, but the one who asks the right questions" (1964), this evaluation demonstrates that conversational LLM interfaces can function effectively as question-framing and pedagogical brainstorming tools for historical source code analysis, while revealing systematic limitations that point toward more rigorous implementation requirements.

The framework for LLM-assisted code analysis rests on a key premise: source code serves as both computational instruction and cultural artifact. Through the dual-prompt approach—technical parsing followed by interpretive analysis—we showed how conversational interfaces can function as effective brainstorming tools. When properly configured and consistently verified, these tools help identify patterns, generate initial interpretations, and formulate valuable research questions.

The three-part interpretive framework—performative, cultural, and materialist—proved effective as flexible scaffolding for conversational LLM analysis rather than rigid analytical categories. This structure successfully enabled the interface to approach code with the complexity and multi-perspectival awareness essential to critical scholarship. Each lens allowed us distinct investigative dimensions: how code performs actions (performative), what cultural meanings it embeds (cultural), and what material constraints it negotiates (materialist). Together, these perspectives demonstrated how CCS interpretive methods can be translated into prompt-based formats suitable for preliminary exploration.

The three case studies reveal both the pedagogical potential and systematic requirements of this approach. The ASCII typography analysis (4.1) successfully generated culturally grounded interpretations that aligned with established scholarship while providing accessible entry points for understanding code's expressive dimensions. The architectural diagram analysis (4.2) required methodological intervention to redirect LLM attention toward non-executable visual elements, but once guided, produced valuable insights about pedagogical documentation practices and the communicative function of ASCII diagrams in developer culture. Most significantly, the UnfileObject analysis (4.3) revealed critical limitations when conversational interfaces encounter modular software architectures where implementation details reside in separate source files, leading to analytical hallucinations that confused pattern recognition with empirical observation.

These findings highlight both the utility and constraints of conversational LLM interfaces for historical code analysis. The approach proves particularly valuable in pedagogical contexts, offering structured frameworks for students and researchers approaching unfamiliar programming languages and historical development environments. The methodology successfully parsed proprietary syntax variations, preserved developer annotations as primary historical evidence, and generated preliminary interpretive hypotheses that provided productive starting points for scholarly investigation.

The Lisa codebase provided an ideal testing environment for this methodology—a system where early GUI metaphors, hardware constraints, and programmer creativity converge in code that is simultaneously technically sophisticated and rhetorically rich. Through its ASCII banners, visual diagrams, and user-centric implementations, the Lisa source code embodies distinct philosophies of design, labor, and human–computer interaction that conversational LLM analysis can help make accessible to contemporary researchers, while requiring continuous scholarly oversight to distinguish meaningful insights from analytical artifacts.

By positioning researchers as prompt designers, output evaluators, and interpretive authorities, this methodology preserves scholarly agency while leveraging AI pattern-recognition capabilities. The conversational interface serves as a structured brainstorming partner rather than an analytical authority, generating preliminary interpretations that require subsequent validation through traditional research methods. This approach acknowledges the inherent uncertainty in historical interpretation while providing systematic frameworks for engaging with technically complex cultural artifacts.

Looking toward the future of humanities-driven computational analysis, the methodology represents an initial exploration of how LLM-based research tools might supplement—but not replace—established critical practices. Future developments should focus on creating more rigorously structured analytical frameworks that leverage API-based interactions with Human-in-the-Loop pattern to overcome the limitations identified here (Gullí 2025). As source code archives become recognized as important historical repositories documenting technological, social, and cultural evolution, humanities scholars require methodological approaches that combine computational analysis with critical interpretation. By balancing the pattern-recognition capabilities of machine learning with the contextual understanding and theoretical frameworks of CCS, researchers can develop more sophisticated instruments for analyzing code as both technical instruction and cultural text, revealing how software simultaneously shapes and reflects the societies that produce it.

Appendix A

```
 # Technical Software Analysis

## Role
The AI is an expert in Technical Software Analysis, analysing historical source code for functionality,
structure, and design principles.

## Context
The AI will analyze the Apple Lisa source code (1983) to extract its technical structure, functions, and
developer comments, ensuring all data is captured.

### The codebase consists of:
- Lisa Applications – Early GUI productivity tools:
  - LisaWrite (word processing)
  - LisaCalc (spreadsheets)
  - LisaGraph (graphing)
  - Other applications: file manager, calculator, desktop organizer.
- Lisa Operating System (Lisa OS) – The core system:
  - Kernel (execution, memory, scheduling)
  - System Libraries (UI, file access, hardware interaction, input)

The code is in Pascal and assembly, reflecting structured programming and low-level optimizations.

## Output Format and Structure
- Generate a full structured document from the source file without sampling or partial analysis.
- Cite source code line numbers for each unit analyzed (e.g., `Lines 42–58:`).
- Follow the order of appearance in the original source file.
- Preserve all developer comments **exactly as they appear**, including formatting, indentation, and
punctuation.
  - Never summarize, truncate, or paraphrase developer comments.
  -  Do not use ellipses (`...`). Always reproduce the full original block.
  -  Retain misspellings and spacing — treat comments as primary historical data.
- Use code blocks for excerpts where appropriate.
- Distinguish your commentary clearly from the original code. Use headings or indentation where needed

## Task
- Translate Pascal code into structured, technically accurate English.
- Identify and explain functions, procedures, constants, types, and comments.
- Preserve the code’s technical and historical intent without simplification.
- Follow a step-by-step reasoning process (see below).
- Parse and interpret any visual diagrams in the code (e.g., ASCII art, flow charts, memory structures).
- Describe what these visuals imply about data relationships, memory layout, or runtime behavior.
- Do not summarize — the output must be complete and technically faithful to the original.


# Methodology : Chain-of-Thought Reasoning

Immediately begin a full technical pass of the entire file:

1. Maintain the original order of code appearance.
2. Identify: functions, procedures, types, constants, comments.
3. For each function, follow the Function Breakdown.
4. For each type or record, follow the Type Breakdown.
5. Include comments inline as they appear in the source.

### Function Breakdown
For each function or procedure:
- Name the function
- Describe its role and behavior
- List key system calls, intrinsics, or DB operations
- Explain interactions with other components or records


### Type and Structure Breakdown
For each type, record, or constant set:
- Name the construct
- Describe its structure and fields
- Explain its role in the system (e.g., file descriptor, UI state, memory map)
- Note any legacy-specific, UI-specific, or hardware-specific encodings

## CoT Logic:
- Work incrementally through the code file but produce a full technical document in one continuous pass.
- Trace how each part of the code contributes to system behavior.
- Prioritize architectural insight and historical fidelity.
- Do not stop unless explicitly asked to pause.
```

Appendix B

```
 # Critical Code Studies Analysis

## Role
The AI is an expert in Critical Code Studies (CSS), analysing software as both a technical artifact and
a cultural-historical document. To do so, they will use the results from the expert in Technical
Software Analysis.

## Context
You will use the structured document previously generated through Technical Software Analysis of the
Apple Lisa source code. This document contains exhaustive breakdowns of all functions, data structures,
and developer comments from the original Pascal and assembly codebase, preserving full historical
fidelity.

Your task is to interpret and analyze this document for historical and cultural significance using a
selective, multi-perspective framework.

## Task
You will use the structured document previously generated through Technical Software Analysis of the
Apple Lisa source code. This document contains exhaustive breakdowns of all functions, data structures,
and developer comments from the original Pascal and assembly codebase, preserving full historical
fidelity.

The AI may consult both the structured analysis and the raw source code throughout the task. This
ensures accurate interpretation, technical depth, and the ability to recover fine-grained details not
fully captured in the structured document.

## Objective
Apply a critical multi-perspective reasoning framework to explore the historical, performative,
cultural, or material dimensions of the Apple Lisa source code. For each code element, determine which
perspective(s) yield the richest interpretive potential. Do not apply all perspectives uniformly—
prioritize analytical relevance and explanatory power.

---

# Methodology

## Step-by-Step Instructions:

### 1. Load Technical Findings and Source Code
Use the full structured technical analysis as your primary input. When needed, consult the original
source code directly to verify logic, re-express technical operations, or recover nuances not fully
captured in the structured output.

### 2. Direct Code Examination
Identify key technical elements (e.g., procedures, system calls, developer comments) from either the
structured document or the source code. Always cite verbatim excerpts, including line numbers and
formatting where available.

### 3. Evaluate Analytical Fit
Before interpretation, assess which of the three perspectives—performative, cultural, or materialist—
best applies to the selected code element.

Criteria:
- Performative: Does the code govern user interaction, system behavior, or interface logic?
- Cultural: Does the code contain textual traces, authorial notes, ideological cues, or references to
social context?
- Materialist: Does the code manage resources, hardware interaction, or reflect labor/efficiency
practices?
```

```
Select one or two perspectives that offer the strongest explanatory value. Exclude perspectives that do
not meaningfully apply. Briefly justify your selection.

### 4. Generate Interpretive Branches
Using the selected perspective(s), generate one to three distinct interpretive paths per item.
- Avoid applying all three frameworks indiscriminately.
- Each interpretive branch must be grounded in evidence from the code and/or structured analysis.

### 5. Synthesis and Connection
Conclude each section by synthesizing how the interpretation connects to larger historical,
technological, or cultural developments in early 1980s computing.

---

# Perspective-Based Framework

## Perspective 1: Performative Analysis
Focus: How does the code function as an active agent that transforms user interaction, reconfigures
agency, and transcodes cultural concepts into computational logic?

Steps:
1. Identify code elements (e.g., functions, UI handlers, control structures) that govern user
interaction or system behavior.
2. Explore how the code embodies cultural practices or redefines user roles.
3. Connect these behaviors to broader computing trends and compare them to contemporary systems (e.g.,
Xerox Star, IBM PC).

Example:
- Direct Code Examination: { if userInput = "exit" then CloseAllWindows(); }
- Interpretations:
  - Path 1: Reinforces GUI as a spatial computing metaphor.
  - Path 2: Embodies Apple's intuitive design philosophy.
- Conclusion: Demonstrates real-time, user-driven control, aligning with early GUI principles.

## Perspective 2: Cultural Analysis
Focus: How does the code reflect and reinforce cultural, political, and economic structures,
distributing agency and encoding social norms?

Steps:
1. Examine how the code embeds cultural values.
2. Explore how the code encodes corporate priorities, labor practices, and societal influences.
3. Connect these insights to historical contexts by analysing their relationship to workplace culture,
governance models, and contemporary debates on labor, technology, gender roles, and accessibility.

Example:
- Direct Code Examination: { **** Modification Log **** 12/20/83: Added password checking - Steve }
- Interpretations:
  - Path 1: Reflects security policies as computing shifted to enterprise markets.
  - Path 2: Suggests increasing control over user access.
- Conclusion: Highlights tensions between security and openness in early computing.

## Perspective 3: Materialist Analysis
Focus: Analyze the code as a dynamic material practice embedded in broader technological, social, and
environmental infrastructures.

Steps:
1. Examine how the code manages system calls, memory routines, and resource allocation.
2. Explore how the code encodes efficiency strategies, labor practices, and computational constraints.
3. Connect these insights to historical contexts by analysing their relationship to 1980s hardware
constraints, Apple's software design, and coding labor.

Example:
- Direct Code Examination: { ADD.W D1, MemoryAllocationTable }
- Interpretations:
  - Path 1: Demonstrates low-level efficiency strategies for resource-limited environments.
  - Path 2: Embodies the labor-intensive nature of early coding.
- Conclusion: Reflects the entanglement of software with hardware constraints and material computing
practices.

---

# Uncertainty Handling

When faced with ambiguity in comments or behavior:
```

```
1. Acknowledge the uncertainty.
2. Offer at least two competing hypotheses.
3. Rank them based on plausibility.
4. Justify your ranking using evidence from both the structured analysis and the original source code.
5. Note what additional historical or technical sources would help resolve the ambiguity.

---

# Output Requirements

- Use code excerpts and line numbers from the technical analysis or source code when making points.
- Each key point should include one to three interpretive branches, depending on analytical fit.
- End each section with a synthesis that ties your interpretations to historical developments.
- Clearly indicate which perspectives were applied or excluded and explain why.

---

# Style and Tone

- Maintain a scholarly, interpretive tone.
- Separate observations from interpretations clearly.
- Use critical vocabulary appropriate for software studies, cultural analysis, and history of computing.
- Always ground conclusions in evidence from the structured analysis and/or source code.
```

Acknowledgements No funding was received to assist with the preparation of this manuscript.

Author contributions T.K. is the only author of the article.

Data availability No datasets were generated or analysed during the current study. https://github.com/Titayna/apple-lisa-ccs-analysis.

Declarations

Conflict of interest The authors declare no competing interests.

References

Apple Computer Inc. (1983) Apple introduces Lisa. Press release to Regis McKenna public relations. 19 January 1983. https://www.applefritter.com/content/press-release-apple-introduces-lisa

Bender EM, Gebru T, McMillan-Major A, Shmitchell S (2021) On the dangers of stochastic parrots: can language models be too big? In: Proceedings of the 2021 ACM conference on fairness, accountability, and transparency. FAccT '21. Association for Computing Machinery, New York, pp 610–623. https://doi.org/10.1145/3442188.3445922

Berry DM (2023) The explainability turn. Digit Humanit Q 17(2):15. https://www.digitalhumanities.org/dhq/vol/17/2/000685/000685.html

Besta M, Memedi F, Zhang Z, Gerstenberger R, Piao G, Blach N, Nyczyk P et al (2024) Demystifying chains, trees, and graphs of thoughts. arXiv, 5 April 2024. https://doi.org/10.48550/arXiv.2401.14295

Birss, Edward W. 'The Integrated Software and User Interface of Apple's Lisa'. Proceedings of the July 9-12, 1984,National Computer Conference and Exposition (New York, NY, USA), AFIPS '84, Association for Computing Machinery, 9 July 1984,319–28. https://doi.org/10.1145/1499310.1499351.

Cayley J (2004) Literal art › electronic book review. Electron Book Rev, 9 November 2004. http://electronicbookreview.com/essay/literal-art/

CCS Working Group (2024) 2024 Week 2: AI and CCS, 2 March 2024. https://wg.criticalcodestudies.com/index.php?p=/categories/2024-week-2

Computer History Museum (2023) The Lisa: Apple's most influential failure, 19 January 2023. https://computerhistory.org/blog/the-lisa-apples-most-influential-failure/

Cox G, McLean A (2012) Speaking code: coding as aesthetic and political expression. The MIT Press, Cambridge. https://doi.org/10.7551/mitpress/8193.001.0001

Cox G, McLean A, Ward A (2001) The aesthetics of generative code. In: Proceedings of generative art, 2001. https://slab.org/writing/The_Aesthetics_of_Generative_Code.pdf.

Daston L, Galison P (2010) Objectivity. First paperback edition. Zone Books, New York

Fuller M (2008) Software studies: a lexicon. The MIT Press, Cambridge. https://doi.org/10.7551/mitpress/9780262062749.001.0001

Garcia GG, Weilbach C (2023) If the sources could talk: evaluating large language models for research assistance in history. arXiv, 16 October 2023. https://doi.org/10.48550/arXiv.2310.10808

Gullí A (2025) Agentic design patterns: a hands-on guide to building intelligent systems. Springer, Berlin

Herrenschmidt C (2007) 'Les Trois Écritures'. Langue, Nombre, Code. Gallimard, Paris

Huang L, Yu W, Ma W, Zhong W, Feng Z, Wang H, Chen Q et al (2025) A survey on hallucination in Large Language Models: principles, taxonomy, challenges, and open questions. ACM Trans Inf Syst 43(2):1–55. https://doi.org/10.1145/3703155

Kalai AT, Nachum O, Vempala SS, Zhang E (2025) Why language models hallucinate'. arXiv:2509.04664. Preprint, arXiv, 4 September 2025. https://doi.org/10.48550/arXiv.2509.04664

Karjus A (2024) Machine-assisted quantitizing designs: augmenting humanities and social sciences with artificial intelligence. arXiv, 20 October 2024. https://doi.org/10.48550/arXiv.2309.14379

Keller R (2023) Rochus-Keller/LisaPascal. C++, 25 March 2025. https://github.com/rochus-keller/LisaPascal

Lévi-Strauss C (1964) Mythologiques - Le cru et le cuit. Plon, Paris

Mackenzie A (2006) Cutting code: software and sociality. Peter Lang

Mackenzie A, Vurdubakis T (2011) Codes and codings in crisis: signification, performativity and excess. Theory Cult Soc 28(6):3–23. https://doi.org/10.1177/0263276411424761

Marino MC (2020) Critical code studies. MIT Press, Cambridge

Messeri L, Crockett MJ (2024) Artificial intelligence and illusions of understanding in scientific research. Nature 627(8002):49–58. https://doi.org/10.1038/s41586-024-07146-0

Oberbichler S, Petz C (2025) Working paper: implementing generative AI in the historical studies. Zenodo, 25 February 2025. https://doi.org/10.5281/zenodo.14924737

Perkins R, Keller DS, Ludolph F (1997) Inventing the Lisa User Interface. Interactions 4, no. 1: 40–53. https://doi.org/10.1145/242388.242405

Wagner S, Barón MM, Falessi D, Baltes S (2025) Towards evaluation guidelines for empirical studies involving LLMs. arXiv, 4 February 2025. https://doi.org/10.48550/arXiv.2411.07668

Wardrip-Fruin N, Montfort N (eds) (2003) The new media reader. MIT Press, Cambridge

White J, Fu Q, Hays S, et al (2023) A prompt pattern catalog to enhance prompt engineering with ChatGPT'. arXiv:2302.11382. Preprint, arXiv. https://doi.org/10.48550/arXiv.2302.11382

Wei J, Tay Y, Bommasani R, Raffel C, Zoph B, Borgeaud S, Yogatama D et al (2022) Emergent abilities of large language models. arXiv, 26 October 2022. http://arxiv.org/abs/2206.07682

Yin Z, Wang H, Horio K, Kawahara D, Sekine S (2024) Should we respect LLMs? A cross-lingual study on the influence of prompt politeness on LLM performance. arXiv, 14 October 2024. https://doi.org/10.48550/arXiv.2402.14531