Lumen: Developer Agency Through Transparent Context Control in AI-Assisted Programming

Nakul Goel
Computer Science Department
Toronto Metropolitan University
Ontario, Canada
nakul.goel@torontomu.ca

Glaucia Melo
Computer Science Department
Toronto Metropolitan University
Ontario, Canada
glaucia@torontomu.ca

***Abstract*—The landscape of software engineering has undergone rapid transformation with the emergence of Artificial Intelligence (AI) coding assistants, which are being increasingly integrated into development workflows to boost productivity and automate routine tasks. Although modern AI coding assistants, such as Claude Code, Cursor, Codex, and many others, can automatically access entire repositories, they operate as black boxes: developers cannot see or control what context drives AI decisions. This automatic operation creates uncertainty about whether critical dependencies, security configurations, or test files are considered. We present Lumen, an open-source tool that takes a fundamentally different approach: preserving developer control through transparent context assembly. Using a double-copy interaction paradigm that leverages natural clipboard behavior, Lumen instantly shows developers the dependency graph and AI-generated summaries of files connected to their copied code. This allows developers to create AI-generated code where the model uses the same context that the developer is focusing on. This design philosophy, which incorporates human-in-the-loop capabilities with full visibility, lays the foundation for future agentic capabilities, ensuring that developers understand and control what the AI sees and does. We demonstrate through Cognitive Walkthrough Analysis how Lumen reduces context assembly overhead while maintaining the transparency necessary for production code. Our open-source implementation provides a foundation for the community to build upon, developing AI tools that enhance rather than replace human judgment.**

***Index Terms*—Software engineering, AI-assisted development, Developer tools, Context-aware systems, Human-computer interaction, Transparent AI**

I. Introduction

The landscape of Artificial Intelligence(AI)-assisted software development has undergone rapid evolution. Tools like GitHub Copilot provide real-time suggestions within IDEs, while conversational systems like ChatGPT and Claude offer sophisticated reasoning about code. More recently, agentic tools such as Claude Code, Cursor, and Aider have emerged, offering the ability to automatically access and analyze entire repositories. These advances promise to transform how developers work, yet introduce a fundamental challenge: the loss of visibility and control over what context drives AI decisions. Recent studies have shown that AI assistance can reduce the productivity of senior developers [1], with experienced developers taking 19% longer to complete tasks when using AI tools compared to working without them.

Consider a developer modifying an authentication endpoint. An AI assistant with automatic repository access might generate functionally correct code that bypasses critical rate-limiting middleware or ignores session validation requirements. The developer cannot verify whether the AI considered these dependencies until the code fails during testing or in an edge case in production. This opacity creates a trust deficit that limits AI adoption for production-critical code, indicating why the productivity of senior developers has decreased despite the increasing capability of AI programming agents.

The problem is not unique to any particular tool. Whether using ChatGPT (requiring manual copy-paste), Cursor (with automatic file access), or Claude Code (with repository-wide analysis), developers face the same challenge: they cannot see or control what context informs AI suggestions. Researchers have found that developers spend significant time verifying AI suggestions and crafting prompts to provide adequate context, time that could be spent on actual development [2].

This context assembly challenge is compounded by workflow disruption. Studies on flow state interruption in software development [3] demonstrate that interruptions during software development cause cognitive recovery periods. Current AI tools, whether manual or automated, often disrupt developers' natural workflow, either through tedious copy-paste operations or by removing them from the decision-making loop entirely.

In this paper, we present Lumen, a tool that aims to address these challenges through a fundamentally different approach: developer-controlled AI assistance with transparent context assembly. When developers copy their code twice during their normal workflow, if Lumen has access to the development project files, Lumen instantly displays the dependency graph and AI-generated summaries of connected files. More importantly, it allows users to add the copied and linked information automatically to the context for a future query. Lumen uses the copy (Ctrl+C) feature to operate. A single copy preserves regular clipboard operation, while a double copy within a second signals intent for AI assistance, keeping Lumen's interface active for context control. In a Cognitive Walkthrough Analysis [4], we found that Lumen's double-copy interaction design significantly reduces the viscosity of context assembly from 15-20 manual operations (in traditional

approaches) to 2-5 guided selections, while maintaining high visibility of dependencies through the visual graph display. Lumen's double-copy interaction design also eliminates premature commitment by allowing developers to preview and adjust context before AI processing.

This design philosophy, which keeps humans in the driver's seat with full visibility, serves two critical purposes. First, it ensures that developers understand what context the AI needs and receives, thereby building trust through transparency. Second, it creates a foundation for future agentic capabilities, where AI can perform more sophisticated tasks while developers maintain oversight and control. This leads us to the central research question guiding this work: **How can we design AI-powered developer tools that enhance transparency and preserve agency without disrupting natural workflows?** The key insight is that effective AI assistance does not require removing human judgment; instead, it requires augmenting human capabilities with transparent, controllable tools.

Our contributions include:

- A novel double-copy interaction paradigm that preserves normal workflow while enabling AI assistance on multiple applications and operating systems
- Transparent context visualization showing file dependencies and AI summaries before processing
- An open-source implementation demonstrating how developer control can coexist with powerful AI capabilities

In this work, we contribute both novel interaction techniques and engineering artifacts. Specifically, the double-copy interaction paradigm is a novel, lightweight approach to triggering AI assistance that preserves developer flow, a design that, to our knowledge, has not been explored in prior developer-AI tooling. In parallel, we present a well-engineered open-source implementation that integrates transparent context visualization, dependency-aware summaries, and customizable AI backends. This separation is key: while prior tools assemble context or visualize dependencies, none provide the combined affordance of interactive, developer-controlled, live context assembly activated from within natural workflows. Lumen's approach addresses current limitations in AI-assisted development while laying the groundwork for improved human-AI collaboration.

II. Related Work

Recent advances in AI-powered tools have significantly reshaped how developers interact with software engineering environments. However, critical challenges persist regarding trust, control, and transparency in integrating these assistants into real-world workflows. Below, we summarize the most relevant literature in two key areas: AI-assisted software development and trust/control in developer-AI interaction.

A. AI-Assisted Software Development

AI-assisted development tools have progressed from single-file completion models, like GitHub Copilot [5], to more advanced systems capable of analyzing entire repositories. Tools such as ChatGPT offer robust reasoning capabilities but disrupt developer workflows by requiring external interaction. Newer systems like Cursor and Claude CLI aim to embed AI directly into developer environments with project-level awareness.

Despite these advancements, developers often face a lack of real-time contextual alignment. Hartman et al. propose a two-stage retrieval mechanism to achieve comprehensive codebase understanding, yet their approach remains challenging to integrate into everyday workflows [6]. Empirical studies, such as Sergeyuk et al.'s survey of 481 programmers, confirm that the absence of project-scale context is a key limitation of current AI assistants [7].

B. Trust and Control in Developer-AI Interaction

A growing body of research highlights how current tools erode developer trust and control. Barke et al. reveal a dual interaction mode, where developers either maintain or lose control depending on the AI's behavior [8].

Large-scale studies, like Liang et al.'s survey at ICSE 2024 with 410 developers, validate these concerns, pinpointing a lack of contextual understanding as a barrier to adoption [9]. While prior work has demonstrated the potential of repository-wide AI assistance and highlighted the challenges surrounding trust and control, to the best of our knowledge, none have successfully integrated transparent, developer-controlled context assembly directly into the natural development workflow. This is the gap our work aims to fill.

III. The Lumen Approach

AI coding assistants are rapidly evolving, but their growing capabilities often come at the cost of developer agency. Existing tools operate either through opaque automation or require intrusive workflow changes, leaving developers uncertain about the context being used and how decisions are made. Lumen takes a different path. It is grounded in the belief that effective AI assistance should amplify, not replace, human judgment. In this section, we present the core principles and interaction mechanisms behind Lumen, designed to offer transparency, context control, and seamless integration with natural developer behavior.

A. Novelty and Design Contribution

Lumen introduces two core innovations that distinguish it from prior work. First, the double-copy interaction paradigm offers a novel trigger mechanism for AI assistance that aligns with existing developer behavior (copy-paste), eliminating the need for explicit commands or context switching. Unlike prior tools that either rely on full automation or manual prompting, this method balances control and flow continuity. Second, Lumen integrates transparent real-time context assembly with a dependency-aware visualization and AI-generated summaries, allowing developers to audit and modify what the AI sees before query submission. While previous tools have implemented some of these capabilities in isolation, Lumen is, to our knowledge, the first to unify them into a cohesive, extensible framework grounded in transparency and human-in-the-loop control.

B. Design Philosophy: Augment, Do not Automate

Lumen's fundamental design principle is that AI should augment developer capabilities rather than automate developer decisions. This philosophy emerges from the recognition that production code requires not just functional correctness but also adherence to security policies, performance requirements, architectural patterns, and team conventions, constraints that are difficult to encode in AI systems but obvious to developers familiar with the codebase.

Unlike automatic tools that operate as black boxes, Lumen ensures every AI interaction is transparent and controllable. Developers see exactly which files will provide context, understand why those files are relevant through AI-generated summaries, and can adjust the selection based on their specific task. This approach transforms AI from an autonomous agent making decisions in isolation to a powerful assistant operating under developer guidance.

The augmentation philosophy extends to workflow integration. Rather than requiring developers to learn new commands or switch to dedicated interfaces, Lumen leverages the universal copy-paste interaction. This design choice reflects our belief that the best tools are those that enhance existing workflows rather than replacing them.

C. The Double-Copy Interaction Paradigm

The double-copy paradigm embodies Lumen's core features, balancing normal workflow with AI assistance.

Single Copy: When a developer copies text, Lumen performs rapid file detection and displays a brief popup showing the identified source file. This pop-up (presented in Figure 1 disappears after one second (configurable in settings), allowing normal clipboard operation to continue. The brief appearance provides ambient awareness without disruption. Developers are aware that Lumen has detected their context, but they are not required to engage with it.

Double Copy: If the developer copies again within one second, Lumen interprets this as intent for AI assistance. The pop-up remains active for 60 seconds, transforming from a simple notification into a comprehensive context control interface. This extended view shows the dependency graph, file summaries, and context selection options. This design serves multiple purposes:

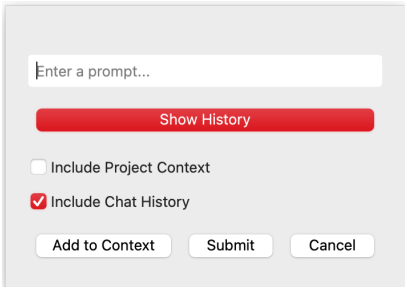

Fig. 1. Double-Copy Interaction Popup Interface with Context Preview.

1) **Reduced Learning Curve**: Every developer knows how to copy; the double-copy is a natural extension
2) **Preserves Normal Workflow**: Single copies work exactly as expected
3) **Intentional Activation**: AI assistance requires explicit action, preventing accidental activation
4) **Immediate Access**: No need to switch applications or invoke special commands

The timing parameters (a 1-second initial timeout and a 60-second extended timeout) were chosen based on informal testing to strike a balance between responsiveness and stability. Users can adjust these values to match their preferences.

D. Transparent Context Assembly

When activated through double-copy, Lumen presents a comprehensive view of available context, as presented in Figure 2. Below we describe the details of this screen.

File Detection and Display: The primary panel shows the detected source file with syntax highlighting. If detection fails (which happens with heavily modified code), developers can manually select the correct file from the project tree. This fallback ensures the tool remains useful even when automatic detection fails.

Dependency Visualization: An interactive graph displays file relationships. Nodes represent files, colored by programming language. Edges show import/export relationships. Node size indicates importance (number of connections). Clicking a node highlights its direct dependencies, making it easy to understand the project structure at a glance.

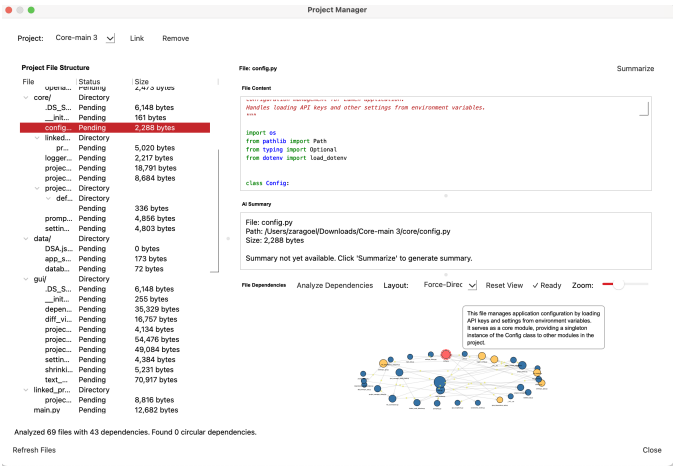

Fig. 2. Interactive Dependency Visualization Graph Showing File Relationships and Project Architecture.

AI-Generated Summaries: Each file in the project has an AI-generated summary explaining its purpose and key functionality. These summaries appear when you hover over the dependency graph and in the file selection list. For example:

- auth-middleware.js: Validates JWT tokens and checks user permissions
- rate-limiter.js: Implements sliding window rate limiting for API endpoints

- user.model.js: Defines User schema with password hashing and validation

Context Selection Interface: A checkbox list shows all project files with clear indicators:

- Automatically selected files (the source file and its direct imports)
- Suggested additions (test files, configurations)
- Available files (all other project files)

Developers can include or exclude any file with a single click. The interface shows the total context size and warns if approaching AI model token limits.

Context Preview: Before sending to AI, developers see exactly what will be transmitted:

```
Selected Context (5 files, ~2000 tokens):
- /src/api/login.js (source file)
- /src/middleware/auth.js (imported by source)
- /src/middleware/rate-limit.js (imported by source)
- /src/config/security.json (suggested - security
    config)
- /test/api/login.test.js (suggested - test file)

Your query: [text input field]
```

This transparency ensures developers know precisely what information the AI will use to generate suggestions.

E. Enabling Developer Control: Human-in-the-Loop

Lumen's design reinforces developer control with a set of features described below.

Explicit Context Boundaries: Unlike automatic tools that might analyze arbitrary files, Lumen only includes files explicitly selected by the developer. This prevents AI from making assumptions based on unrelated code or outdated files.

Query Refinement: The text input field allows developers to provide specific instructions along with the code context. For example: "Fix this login function but maintain backward compatibility with the v1 API". This ensures that AI suggestions align with requirements that may not be evident from the code alone.

F. Integration with Development Workflow

Lumen integrates seamlessly with existing development environments by integrating the following features:

Editor Agnostic: By operating at the clipboard level, Lumen works with any editor or IDE. Developers do not need to switch tools or learn new plugins.

Project Awareness: Once linked to a project directory, Lumen maintains awareness of file structure, recent changes, and dependencies. This persistent context reduces setup overhead.

AI Provider Flexibility: Developers can choose their preferred AI backend:

- Claude for complex reasoning and large context windows
- GPT-4 for general-purpose assistance
- Groq for rapid responses
- Local models for privacy-sensitive code

This flexible integration ensures Lumen enhances existing workflows rather than requiring wholesale process changes.

IV. Implementation

To realize Lumen's vision of transparent, developer-controlled AI assistance, we designed and built a modular system optimized for responsiveness, extensibility, and workflow compatibility. This section details the major components of our implementation. We begin with the system architecture, describing how Lumen leverages Qt's event-driven model to achieve a responsive UI. We then explain the file detection algorithm used to map copied code to source files, followed by our comprehensive dependency analysis and AI-powered summary generation. Next, we describe the context assembly pipeline that intelligently prepares AI input. We conclude by highlighting key performance optimizations and discussing the trade-offs inherent in our design decisions.

A. System Architecture

A version of Lumen built using Python libraries can be found at https://github.com/lumenEngines/Core.

Below we discuss an event-driven architecture that maintains responsiveness while handling complex operations.

Main Application Thread: The PyQt5 application runs in a single main thread using Qt's event loop. The main entry point (`main.py`) initializes:

- `TextWindow` class for the primary UI
- Multiple API connections (Anthropic, Groq)
- Screenshot worker thread for visual input processing
- Project management and file tracking systems

Clipboard Monitoring: Rather than a separate thread, Lumen uses Qt's `QTimer` for efficient clipboard polling:

- Configurable polling interval (default from settings)
- Detects changes through content comparison
- Implements double-copy detection for intentional triggers
- Runs within the main event loop to avoid threading complexity

Asynchronous Processing: Computationally intensive tasks use `ThreadPoolExecutor`:

- File summarization with configurable worker threads
- Batch processing for multiple files
- Dependency analysis operations
- API calls with timeout handling

UI Components: The interface includes:

- Web-based content display using `QWebEngineView`
- Interactive dependency graph visualization with force-directed layout
- Project manager dialog for file organization
- Settings dialog for configuration

B. File Detection Algorithm

Lumen implements a multi-strategy approach for sub-millisecond file detection.

Strategy 1: Hash-Based Line Matching (95% confidence). The system maintains MD5 hashes of all meaningful lines for O(1) lookup time complexity.

Strategy 2: Word Sequence Matching. Creates sliding windows of word sequences for robust partial matching.

Strategy 3: Identifier Indexing. This feature is used to index and track identifiers (such as function names, variables, and class names) across files in a codebase.

Strategy 4: Sliding Window Fuzzy Matching. The `SmartContextMatcher` offers enhanced fuzzy matching capabilities.

C. Dependency Analysis

Lumen's dependency analyzer (`FileDependencyAnalyzer`) supports 40+ programming languages through pattern-based parsing. The dependency analyzer is designed to understand how different files in a project relate to each other, specifically, which files import or depend on others. This is crucial for accurately identifying context during AI-assisted development. Lumen uses pattern-based parsing to detect import statements within source code. These are language-specific regular expressions (regex patterns) that match how dependencies are declared in different programming languages.

For example: In Python, you might write import utils or from auth import login. In JavaScript, it could be import login from './auth' or const db = require('./db'). Lumen defines and uses regular expressions tailored to each language to extract these relationships.

```
# Language-specific import patterns
self.import_patterns = {
    'python': [
        r'^\s*from\s+([^\s]+)\s+import',
        r'^\s*import\s+([^\s,]+)'
    ],
    'javascript': [
        r'import\s+.*\s+from\s+[\'"]([^\'"\)]+)[\'"]',
        r'require\s*\(\s*[\'"]([^\'"\)]+)[\'"]',
        r'export\s+\*\s+from\s+[\'"]([^\'"\)]+)[\'"]'
    ],
    'typescript': [
        # Same as JavaScript plus type imports
        r'import\s+type\s+.*\s+from\s+[\'"]([^\'"\)]+)[\'"]'
    ],
    # ... patterns for 40+ languages
}
```

To model the structure of software projects and their inter-file relationships, Lumen's dependency analyzer constructs a directed graph using the NetworkX library. In this graph, each node represents a source file, and each directed edge indicates a dependency, such as an import or module usage, from one file to another. This representation is critical for accurately capturing the flow of dependencies within a codebase and serves as the foundation for several core features of Lumen, including interactive visualizations, context suggestion, and impact analysis. By leveraging this graph structure, Lumen enables developers to trace how changes in one file may affect others, identify missing or hidden dependencies, and reason about the system's architecture as a whole. This design not only supports transparent and informed AI interactions but also reduces cognitive load by externalizing the codebase's structure, allowing developers to navigate and understand complex systems more efficiently.

```
def analyze_project(self, project_root, file_list=None):
    self.graph = nx.DiGraph()

    for file_path in file_list:
        file_info = self.analyze_file(file_path, project_root)
        if file_info:
            self.graph.add_node(file_path, **file_info)

            # Add edges for dependencies
            for dep in file_info.get('imports', []):
                resolved_path = self._resolve_import_path(
                    file_path, dep['module'], project_root
                )
                if resolved_path:
                    self.graph.add_edge(file_path, resolved_path)
```

Additionally, Lumen includes an interactive visualization widget with:

- Force-directed graph layout
- Flowing particle animations showing dependency direction
- Language-based color coding
- Click interactions for exploring dependencies

D. AI Summary Generation

File summaries are generated through a `FileSummarizer` pipeline.

```
def summarize_file(self, file_path, file_content, file_list):
    # Read up to 100KB of file content
    content_to_summarize = file_content[:102400]

    prompt = self._create_summary_prompt(
        file_path, content_to_summarize, file_list
    )

    # Try Anthropic API first, fall back to Groq
    try:
        summary = self.api_manager.call_anthropic_api(prompt)
    except:
        summary = self.api_manager.call_groq_api(prompt)

    # Cache the summary
    self._save_summary_to_cache(file_path, summary)
    return summary
```

Key features:

- Processes files up to 100KB
- Structured prompts requesting specific information
- Dual API support with fallback
- Persistent JSON-based caching
- Batch processing with rate limiting

E. Context Assembly Pipeline

The context assembly in `main.py` follows a multi-source approach:

```
def callCompletionAPI(searchText):
    context_parts = []

    # 1. Accumulated context buffer
    if hasattr(textWindow, 'context_buffer') and
    textWindow.context_buffer:
        context_parts.append("Accumulated context:\n
    " + textWindow.context_buffer)

    # 2. Selected file context with full content
    if selected_file_path:
        with open(selected_file_path, 'r') as f:
            file_content = f.read()
        file_context = f"File: {file_name}\n```\n{
    file_content}\n```"
        context_parts.append(file_context)

    # 3. Smart context matching
    include_context, context_type, context_content =
     \
        smart_context_manager.should_include_context
    (searchText, user_requested)
    if include_context:
        context_parts.append(context_content)

    # 4. Combine with user query
    full_text = "\n\n".join(context_parts) + "\n\
    nCurrent request:\n" + searchText
```

F. Performance Optimizations

To handle large projects efficiently, Lumen implements several optimizations, as presented below.

Hash-Based Indexing: Pre-computed lookup tables enable sub-millisecond file detection:

```
# Truncated hashes for memory efficiency
line_hash = hashlib.md5(line.encode()).hexdigest()
    [:12]  # 12 chars
seq_hash = hashlib.md5(sequence.encode()).hexdigest
    ()[:10]  # 10 chars
```

Smart File Filtering: Excludes problematic directories and large files:

```
EXCLUDED_DIRS = {
    'node_modules', '.git', '__pycache__', '.
    pytest_cache',
    'venv', 'env', 'build', 'dist'
}

# Skip large files for instant detection
if os.path.getsize(file_path) > 500 * 1024:  # 500KB
     limit
    continue
```

Caching Strategies:

- API response caching with content-based keys
- Project summary caching in JSON files
- Lazy loading of project files
- In-memory normalized file content cache

Parallel Processing: For ultimate performance using multi-core architectures.

```
with ThreadPoolExecutor(max_workers=self.max_workers
    ) as executor:
    # Process in controlled batches
    batch_size = min(self.max_workers, 5)
    for i in range(0, len(files), batch_size):
        batch = files[i:i + batch_size]
        futures = [executor.submit(self.
    summarize_file, f) for f in batch]
```

G. Limitations and Trade-offs

The current implementation has several limitations:

No True Multi-threading for UI: Uses Qt's event loop rather than separate threads, trading some parallelism for simplicity and stability.

Static Analysis Only: The dependency analyzer cannot detect:

- Runtime imports (`__import__`, `importlib`)
- Conditional imports
- String-based dynamic imports

Memory Considerations:

- 500KB limit for instant file detection
- 1MB limit for context matching
- In-memory caching may consume significant RAM for large projects

API Rate Limiting:

- 0.5-second minimum between API calls
- Batch size limits to prevent overwhelming APIs
- No streaming support for real-time responses

These trade-offs prioritize reliability, maintainability, and broad compatibility. The modular architecture allows teams to extend functionality for specific use cases while maintaining the core vision of transparent, developer-controlled AI assistance.

V. Evaluation

To evaluate Lumen's impact on developer experience, we conducted a cognitive walkthrough analysis grounded in the Cognitive Dimensions of Notations framework [10], [11] and design inspection methods by Wharton et al. [12]. This analytical approach enabled us to systematically evaluate how Lumen's design choices impact key aspects of developer cognition and workflow.

Our evaluation proceeded in three stages. First, we analyzed four relevant cognitive dimensions: viscosity, visibility, premature commitment, and hidden dependencies, comparing Lumen against traditional copy-paste workflows and automatic repository-aware AI assistants. Next, we applied these comparisons to three representative development scenarios: debugging a login failure, adding rate limiting, and refactoring authentication logic. Finally, we synthesized the findings to understand Lumen's overall impact on developer productivity and workflow, while acknowledging the limitations of our evaluation methodology.

A. Evaluation Method

Our evaluation focused on four cognitive dimensions particularly relevant to developer tools:

- **Viscosity**: Resistance to change in the environment
- **Visibility**: Ability to view components readily
- **Premature Commitment**: Requirements to make decisions before necessary information is available
- **Hidden Dependencies**: Important links between entities that are not visible

We analyzed two representative scenarios:

1) **Bug Fix**: Debugging a login failure requiring cross-file investigation
2) **Feature Addition**: Adding rate limiting to multiple API endpoints

Each scenario was evaluated using three approaches:

- Traditional: Manual copy-paste to external AI (ChatGPT/Claude)
- Automatic: Using repository-aware AI (Cursor/Claude Code)
- Lumen: Our transparent context assembly approach

Table I presents the results of the evaluation.

TABLE I
COGNITIVE DIMENSION ANALYSIS

Viscosity (Resistance to Change)	
Traditional	High: Requires 15-20 manual operations including file navigation and copy-paste cycles
Automatic	Low viscosity but inflexible: Single command execution without adjustment capability
Lumen	Low viscosity with control: 2-5 guided operations via double-copy interaction
Visibility	
Traditional	Poor: File relationships remain opaque; dependencies easily overlooked
Automatic	None: Black-box operation prevents verification of analyzed files
Lumen	High: Interactive dependency graph with AI summaries and context preview
Premature Commitment	
Traditional	Low commitment but high effort: Iterative refinement requires complete cycles
Automatic	High: Developers must trust initial results without preview capability
Lumen	Deferred: Preview and adjustment before submission eliminates exploration penalty
Hidden Dependencies	
Traditional	Hidden: Developers must maintain mental models of file relationships
Automatic	Operationally hidden: No indication of which dependencies AI considers
Lumen	Explicit: Visual graph reveals some important relationships with highlighted suggestions

B. Scenario Walkthroughs

We selected to analyze these scenarios (bug fix and feature addition) because they demonstrate different aspects of context assembly challenges, from discovering hidden dependencies during debugging, to maintaining consistency across multiple endpoints, to preserving functionality during structural changes. Each scenario requires understanding relationships between multiple files, making them ideal for comparing how different approaches handle context transparency and control.

Scenario: Debugging Login Failure

Traditional Approach: Developer receives bug report about login failures. They:

1) Open login endpoint file
2) Copy relevant function
3) Paste into ChatGPT
4) Receive suggestion about validation
5) Realize they forgot auth middleware
6) Navigate to middleware file
7) Copy middleware code
8) Paste into ongoing chat
9) Receive new suggestion considering middleware
10) Still miss rate limiter configuration

Time: 15 minutes
Context switches: 8
Files missed: 2 (rate limiter, tests)

Lumen Approach:

1) Copy problematic login code
2) Double-copy to activate Lumen
3) See dependency graph showing auth middleware, rate limiter
4) Notice rate limiter config highlighted in red (modified recently)
5) Include all relevant files with 3 clicks
6) Ask "Why might logins fail intermittently?"
7) AI identifies rate limiter misconfiguration

Time: 3 minutes
Context switches: 1
Files missed: 0

VI. REAL-WORLD USAGE EXAMPLES

To illustrate Lumen's practical benefits and how it addresses real development challenges, we present a series of real-world scenarios drawn from typical software engineering workflows. These examples demonstrate how Lumen enhances visibility, reduces cognitive effort, and prevents common issues caused by opaque AI assistance. Each scenario contrasts development with and without Lumen, showcasing the tool's impact on debugging, security compliance, and team alignment.

A. Understanding Unfamiliar Code

Scenario: A developer joins a team working on an e-commerce platform. They are tasked with fixing a bug in the checkout process, but have never seen the codebase before.

Without Lumen: The developer opens the checkout controller and sees a function call to `calculateShipping()`. They grep for this function, finding it in `/services/shipping.js`. Opening that file, they see it imports from `../models/order` and `../utils/geo`. They open these files, trying to build a mental model. After 20 minutes of exploration, they still have not found the shipping rate configuration that's causing the bug.

With Lumen: The developer copies the `calculateShipping()` call and double-taps Ctrl+C. Lumen instantly shows:

- The shipping service file with summary: "Calculates shipping costs based on weight, distance, and shipping method"

- A dependency graph revealing connections to:
  - `/config/shipping-rates.json` (highlighted in red - recently modified)
  - `/models/order.js` - "Order model with address validation"
  - `/utils/geo.js` - "Geographic distance calculations"
  - `/tests/shipping.test.js` - "Tests for shipping calculations"

The developer immediately sees the configuration file was changed yesterday. They include all files in the context and ask the AI: "The shipping calculation is returning incorrect values for international orders. What might be wrong?"

The AI, seeing both the code and configuration, identifies that the new configuration uses metric units, while the code expects imperial units. A bug is found in 3 minutes, instead of an hour of exploration.

B. Ensuring Security Compliance

Scenario: A developer needs to add a new endpoint for password reset functionality. The company has strict security requirements that all endpoints must implement rate limiting, authentication checks, and audit logging.

Without Lumen: The developer copies an existing endpoint as a template and asks AI to help create the password reset functionality. The AI generates clean, working code that handles password reset perfectly. During code review, security team flags three issues:

- No rate limiting applied
- Audit logging missing
- Password complexity validation inconsistent with company standards

The developer has to refactor, adding dependencies they did not know existed.

With Lumen: The developer copies the template endpoint. Lumen shows the dependency graph with several middleware files highlighted:

- `/middleware/rate-limit.js` - Sliding window rate limiter
- `/middleware/audit.js` - Logs all sensitive operations
- `/config/security.json` - Security policies and password rules
- `/validators/password.js` - Password complexity validation

The warnings indicate these are commonly used with endpoints but were not in the copied code. The developer includes all security-related files and asks: "Create a password reset endpoint that follows all our security requirements." The AI, seeing the security middleware and configuration, generates code that:

- Applies rate limiting using the existing middleware
- Includes audit logging for password reset attempts
- Uses the company's password validation rules
- Follows the established security patterns

Code review passes without security issues.

C. Learning Team Patterns

Scenario: A new developer needs to add a feature for bulk order processing. The team has established patterns for batch operations, but these are not documented.

With Lumen: The developer copies code from a single order processing function. Lumen reveals a pattern through the dependency graph:

- Multiple files with "batch" in their names connect to a central `/utils/batch-processor.js`
- These files all import from `/queues/job-queue.js`
- They share error handling through `/utils/batch-errors.js`
- Each has corresponding files in `/monitoring/` for batch job tracking

Without asking anyone, the developer observes the team's pattern: batch operations utilize a job queue, share error handling, and necessitate monitoring setup. They include examples of other batch operations and ask the AI: "Following the team's batch processing pattern, how should I implement bulk order processing?"

The AI recognizes the established patterns and generates code that fits seamlessly with team conventions. These real-world examples illustrate how Lumen's transparency prevents common development pitfalls:

- Missing dependencies lead to bugs
- Security requirements get overlooked
- Refactoring breaks edge cases
- Team patterns are not followed
- Production environments differ from development

By making context visible and adjustable, Lumen helps developers catch these issues before they become problems.

VII. Discussion

A. The Case for Developer Control

The evolution of AI coding assistants reveals a fundamental tension between capability and control. As tools become more powerful, evolving from single-line completion to repository-wide analysis, developers paradoxically lose visibility into their operation. This opacity creates anxiety, particularly for production systems where a missed dependency or ignored security constraint can have serious consequences [13], [14].

Lumen's approach demonstrates that this trade-off is unnecessary. By making context assembly transparent and keeping developers in control, we can leverage AI's capabilities while maintaining the human judgment essential for quality software. The key insight is that developers do not need AI to make decisions for them; they need AI to help them make better decisions with complete information.

This philosophy becomes even more crucial as AI capabilities continue to expand. Future systems will likely be able to implement complex features, refactor entire codebases, and autonomously fix bugs. Without transparency and control, these powerful capabilities become liabilities; developers may not trust suggestions they cannot verify or understand [8].

B. Transparency as a Foundation for Trust

Trust in AI systems is not built solely through impressive capabilities; it requires an understanding of how those capabilities are applied. When developers can see exactly what context informs AI suggestions, they can:

1) **Verify Completeness**: Ensure all relevant files are considered
2) **Understand Reasoning**: See why AI made specific suggestions
3) **Learn Patterns**: Discover codebase relationships they did not know
4) **Prevent Errors**: Catch missing dependencies before they cause problems

This transparency transforms AI from a mysterious oracle to an understandable assistant. Recent surveys [7], [9] indicate that developers report feeling more confident using AI when they can see its inputs, even if the AI's internal reasoning remains opaque.

C. Workflow Integration and Adoption

One of the most significant barriers to adopting AI tools is workflow disruption. Developers have established patterns for navigating code, debugging issues, and implementing features. Tools that require abandoning these patterns face resistance, regardless of their capabilities [15].

Lumen's clipboard-based approach succeeds because it enhances existing behavior rather than replacing it. Every developer copies code for searching, sharing, or reference. By building on this universal action, Lumen requires minimal behavior change while providing maximum value.

The double-copy paradigm specifically addresses the balance between availability and intrusion. Single copies remain unaffected, allowing for a normal workflow. The intentional second copy signals desire for assistance without requiring new keyboard shortcuts or commands to remember.

D. Open Source and Community Development

Lumen's open-source foundation is central to its design philosophy, enabling diverse teams to adapt the tool to their specific needs and workflows. For example, security-focused teams may extend Lumen with validators to prevent sensitive files from being sent to external AI providers, while large organizations might integrate the tool with internal documentation systems or proprietary knowledge bases. In specialized domains, teams can add support for custom programming languages or framework-specific heuristics, and privacy-conscious developers may choose to reconfigure Lumen to run entirely with local AI models. Rather than offering a rigid, one-size-fits-all solution, Lumen provides a flexible foundation that embraces extensibility as a core principle. Its architecture supports modular customization through pluggable file analyzers, configurable AI backends, hookable context assembly pipelines, and a modifiable UI that can be tailored to fit team-specific development workflows. This design empowers the community to evolve Lumen in alignment with their unique requirements, reinforcing its value as a collaborative, adaptable platform.

E. Limitations

While Lumen demonstrates the value of transparent context assembly, several limitations point to future research directions, which we discuss below.

Dynamic Behavior Understanding: Current static analysis cannot capture runtime dependencies, configuration loading, or conditional imports. Future versions might integrate with runtime analysis tools to provide complete dependency information.

Semantic Understanding: The system currently treats all files equally, but some files are more relevant than others for specific tasks. Machine learning approaches could help prioritize files based on task context.

Cross-Repository Dependencies: Modern development often spans multiple repositories. Extending Lumen to handle microservices or library dependencies would increase its utility for enterprise development [16].

Collaborative Context: Teams have shared knowledge about code that is not captured in files. Integrating with documentation, commit messages, or team communications could provide richer context.

Performance at Scale: While adequate for most projects, very large codebases (millions of files) challenge current indexing approaches. Distributed indexing or incremental analysis could address these limitations.

Empirical Validation Through User Studies: While this paper presents a design-oriented evaluation using cognitive walkthroughs, a key limitation is the absence of an empirical user study. To fully assess Lumen's impact on usability, productivity, and trust in AI assistance, we plan to conduct structured studies with professional developers and software engineering students. These studies will combine observational protocols, think-aloud methods, and post-task interviews to triangulate the effectiveness of Lumen's interaction model in real-world workflows. Future work will also explore longitudinal deployments to assess adoption, learning curves, and tool stickiness over time.

F. Implications for AI-Assisted Development

Lumen's approach has broader implications for how we think about AI in software development:

Human-AI Collaboration: Rather than pursuing fully autonomous AI systems, we should focus on tools that amplify human capabilities. The most effective AI assistants will be those that work with developers, not instead of them [17].

Explainable AI Operations: As AI becomes more prevalent in development tools, explaining not just what AI suggests but what information informed those suggestions becomes critical for adoption and trust [6].

Contextual Awareness: The value of AI suggestions is directly proportional to the quality and completeness of context. The tools must help developers provide rich context without excessive manual effort.

Gradual Automation: Instead of immediately jumping to full automation, tools should provide a spectrum of assistance levels. Developers can start with transparent assistance and gradually allow more automation as trust builds.

VIII. Conclusion

AI coding assistants are transforming software development, but their effectiveness is constrained by a lack of transparency and developer control over the context in which these systems operate. Lumen was motivated by this growing tension between automation and trust. By introducing a double-copy interaction paradigm and transparent context assembly, Lumen empowers developers to direct AI assistance without disrupting their workflow. Through a cognitive walk-through analysis, we demonstrated that Lumen significantly reduces cognitive load and preserves flow state while making dependencies visible and adjustable. This design leads to fewer context-switching operations across several development tasks. Our findings support the argument that trustworthy, agentic AI tools must prioritize visibility and developer agency, setting the foundation for a more integrated future between developers and code-generating systems. Future work should address an empirical evaluation of the tool with software practitioners, as well as longitudinal studies to assess tool adoption and scalability.

References

[1] METR, "Measuring the impact of early-2025 ai on experienced open-source developer productivity," July 2025.

[2] S. T. AI, "Not so fast: Ai coding tools can actually reduce productivity," *Second Thoughts AI*, January 2025, developers spent substantial time reviewing AI output, with multiple rounds of prompting, waiting, reviewing, and discarding.

[3] A. N. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann, "Reducing interruptions at work: A large-scale field study of flowlight," in *Proceedings of the 2017 CHI Conference on Human Factors in Computing Systems*. ACM, 2017, pp. 61–72.

[4] C. Lewis, P. Polson, C. Wharton, and J. Rieman, "Testing a walkthrough methodology for theory-based design of walk-up-and-use interfaces," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1990, pp. 235–242.

[5] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. d. O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman *et al.*, "Evaluating large language models trained on code," *arXiv preprint arXiv:2107.03374*, 2021.

[6] J. Hartman, R. Mehrotra, H. Sagtani, D. Cooney, R. Gajdulewicz, B. Liu, J. Tibshirani, and Q. Slack, "Ai-assisted coding with cody: Lessons from context retrieval and evaluation for code recommendations," in *Proceedings of the 18th ACM Conference on Recommender Systems*. ACM, 2024, pp. 553–562.

[7] A. Sergeyuk, Y. Golubev, T. Bryksin, and I. Ahmed, "Using ai-based coding assistants in practice: State of affairs, perceptions, and ways forward," *arXiv preprint arXiv:2406.07765*, 2024.

[8] S. Barke, M. B. James, and N. Polikarpova, "Grounded copilot: How programmers interact with code-generating models," *Proceedings of the ACM on Programming Languages*, vol. 7, no. OOPSLA1, pp. 85–111, 2023.

[9] J. T. Liang, C. Yang, and B. A. Myers, "A large-scale survey on the usability of ai programming assistants: Successes and challenges," in *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. IEEE, 2024, pp. 616–628.

[10] T. R. Green, "Cognitive dimensions of notations," *People and Computers V*, pp. 443–460, 1989.

[11] T. R. Green and A. F. Blackwell, "Cognitive dimensions of notations: Design tools for cognitive technology," University of Cambridge Computer Laboratory, Technical Report, 1998.

[12] C. Wharton, J. Rieman, C. Lewis, and P. Polson, "The cognitive walkthrough method: A practitioner's guide," in *Usability inspection methods*. John Wiley & Sons, 1994, pp. 105–140.

[13] R. Wang, R. Cheng, D. Ford, and T. Zimmermann, "Investigating and designing for trust in ai-powered code generation tools," in *Proceedings of the 2024 ACM Conference on Fairness, Accountability, and Transparency*. ACM, 2024, pp. 1475–1493.

[14] N. Perry, M. Srivastava, D. Kumar, and D. Boneh, "Do users write more insecure code with ai assistants?" in *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2023, pp. 2785–2799.

[15] P. Vaithilingam, T. Zhang, and E. L. Glassman, "Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models," in *CHI Conference on Human Factors in Computing Systems Extended Abstracts*. ACM, 2022, pp. 1–7.

[16] J. D. Weisz, S. Kumar, M. Muller, K.-E. Browne, A. Goldberg, E. Heintze, and S. Bajpai, "Examining the use and impact of an ai code assistant on developer productivity and experience in the enterprise," *arXiv preprint arXiv:2412.06603*, 2024.

[17] H. Mozannar, G. Bansal, A. Fourney, and E. Horvitz, "Reading between the lines: Modeling user behavior and costs in ai-assisted programming," in *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems*. ACM, 2024, pp. 1–14.