**Program Comprehension Research in the Age of AI Code Assistants**

**Introduction and Problem Context**

With the rise of intelligent coding assistants (e.g. GitHub Copilot, ChatGPT), developers can now generate code or receive code explanations on demand. This raises a fundamental question: **does human program comprehension still matter when AI can write and explain code for us?** Recent research indicates that it does – perhaps more than ever – but the nature of comprehension is evolving. While AI assistance often boosts productivity, it also introduces new human cognitive tasks, like validating and understanding AI-generated code[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=While%20some%20research%20shows%20significant,when%20collaborating%20with%20AI%20assistants)[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=significant%20difference%20in%20the%20time,11%5D%20found%20that%20while). In practical terms, *time saved writing code is often offset by time spent reviewing and fixing AI suggestions*, a mental effort not captured by traditional metrics[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=While%20some%20research%20shows%20significant,when%20collaborating%20with%20AI%20assistants). For example, Mozannar et al. (2024) found that although developers perceive AI tools as beneficial, *AI-generated suggestions create new bottlenecks in reading, understanding, and verifying code*[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=significant%20difference%20in%20the%20time,11%5D%20found%20that%20while). Likewise, Imai et al. observed that Copilot’s outputs, while increasing initial code volume, required more follow-up modification than human-written code[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=al.%C2%A0,10). These findings underscore that **human comprehension remains vital**: developers must integrate AI outputs into larger systems, ensure correctness, and maintain code quality, all of which demand deep understanding.

Moreover, overreliance on automation can erode essential skills. Studies of human-AI interaction warn that as automation increases, critical cognitive skills can atrophy, leaving humans less prepared to intervene when the AI errs[microsoft.com](https://www.microsoft.com/en-us/research/wp-content/uploads/2023/12/NFWReport2023_v5.pdf#:~:text=%E2%80%A2%20An%20increase%20in%20automation,Automation%20also). In the context of coding, if developers skip understanding code because “the AI handles it,” they may struggle to debug or adapt that code in novel situations. *Bainbridge’s classic “irony of automation”* holds true: automation can reduce cognitive load in the short term but also diminish the practitioner’s situational awareness and problem-solving abilities when manual intervention is needed[microsoft.com](https://www.microsoft.com/en-us/research/wp-content/uploads/2023/12/NFWReport2023_v5.pdf#:~:text=%E2%80%A2%20An%20increase%20in%20automation,Automation%20also). Thus, recent discussions emphasize that **human program comprehension is still indispensable** – not just for catching AI mistakes, but for sustaining long-term code quality, enabling effective collaboration, and retaining the ability to think critically about software[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=written%20code%20in%20real,powered%20software%20development%20platform)[microsoft.com](https://www.microsoft.com/en-us/research/wp-content/uploads/2023/12/NFWReport2023_v5.pdf#:~:text=%E2%80%A2%20When%20the%20LLM%20made,tools%20are%20integrated%20into%20workflows). In summary, even in the age of AI assistants, *code is ultimately written for humans to read*. Ensuring that developers understand the code – whether written by themselves, others, or an AI – remains a core concern of software engineering research and practice[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=written%20code%20in%20real,powered%20software%20development%20platform)[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=At%20Atlassian%2C%20code%20readability%20has,its%20value%20in%20this%20LLM).

**Cognitive Models: Continuity and Evolution**

Time-tested cognitive models of program comprehension (focused on attention, working memory, and strategy) continue to apply, though they are being reexamined under the lens of human-AI collaboration. **Working memory and attention** are still recognized as limiting factors in understanding code. Brain imaging studies confirm that reading code activates regions related to working memory, attention, and language processing[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=electroencephalography%20%28EEG%29%2C%20while%2036,memory%2C%20attention%2C%20and%20language%20processing), aligning with classical models that a developer builds a mental representation of code in short-term memory. In fact, Siegmund et al. showed distinct neural activation patterns in programmers consistent with the load on working memory and focused attention during comprehension tasks[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=electroencephalography%20%28EEG%29%2C%20while%2036,memory%2C%20attention%2C%20and%20language%20processing). This suggests that despite new tools, the human brain’s capacity to process code has not fundamentally changed – we still face cognitive load when interpreting complex logic or poor code structure.

However, **developers’ strategies and mental models are adapting** in the presence of AI assistance. For instance, validation of AI-generated code might invoke different tactics than traditional coding. Tang et al. observe that developers exhibit *“LLM-specific” comprehension behaviors*: when tasked with debugging AI-written code, they frequently switch between code and its comments, alter their attentional focus, and often choose to delete-and-rewrite code rather than incrementally fix it[nztang.com](https://www.nztang.com/assets/files/papers/tang_vlhcc24.pdf#:~:text=conducted%20semi,These%20findings%20enhance). These behaviors indicate a shift in strategy – developers approach AI-originated code with a mix of skepticism and opportunism, sometimes treating it as a draft to be reworked. Notably, Tang’s study found that simply **being aware** code was AI-generated affected developers’ mental models of the code’s reliability and likely errors, which in turn altered their approach and even increased their cognitive load[nztang.com](https://www.nztang.com/assets/files/papers/tang_vlhcc24.pdf#:~:text=also%20impact%20their%20mental%20models,powered%20code%20generation)[nztang.com](https://www.nztang.com/assets/files/papers/tang_vlhcc24.pdf#:~:text=greater%20visual%20attention%20and%20cognitive,developers%20are%20informed%20that%20the). This awareness effect suggests a redefinition of cognitive models: developers incorporate *provenance* of code into their comprehension process, a factor absent in traditional models. In effect, a programmer’s internal strategy now might include questions like “this code was suggested by an AI – what kinds of mistakes should I watch for?” This meta-cognitive step is a new component of comprehension in the AI era[nztang.com](https://www.nztang.com/assets/files/papers/tang_vlhcc24.pdf#:~:text=its%20provenance%20%28i,Understanding%20these%20differences%20can%20help).

Traditional comprehension theories (e.g. top-down versus bottom-up reading, or **plan-driven** understanding versus opportunistic understanding) are also being revisited. AI assistants encourage a more opportunistic style: developers often jump directly to solutions (AI outputs) and then backfill understanding as needed. Cognitive theories are beginning to account for this by framing the human–AI interaction in terms of **attention investment**. Blackwell’s Attention Investment Model, originally used to explain how programmers decide to expend cognitive effort, is now applied to AI usage. For example, Lee et al. (2024) use this model to describe naming decisions with Copilot: an AI may suggest a *conventional* identifier name at low cognitive cost, but a programmer can choose to invest extra mental effort to devise a more specific, informative name for clarity[advait.org](https://advait.org/files/lee_2024_copilot_predictability.pdf#:~:text=Mixed,attention%20investment%20model%20are%20both). This trade-off between accepting easy AI suggestions and exerting additional effort for potential long-term comprehension benefits is exactly the kind of cognitive decision-making that contemporary models are incorporating. In summary, core human cognitive processes – working memory limits, attention focus, mental models of code – **remain central** to program comprehension research. The difference now is that they are studied not in isolation, but in interaction with AI assistance. Researchers are refining these models to capture how tools reshape (but do not remove) the cognitive steps a developer goes through when understanding code. As one study puts it, *the mental demands of programming persist, but their distribution is shifting* – less effort in typing, perhaps, but more in reading, verifying, and contextualizing[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=While%20some%20research%20shows%20significant,when%20collaborating%20with%20AI%20assistants)[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=Despite%20growing%20evidence%20that%20AI,unreliable%20for%20assessing%20cognitive%20load).

**Empirical Methods for Measuring Comprehension and Cognitive Effort**

Modern program comprehension research employs a rich toolbox of empirical methods – often combining classic human-factors measures with newer physiological and behavioral data – to quantify how much mental effort developers expend. **Self-reported measures** remain common: for example, the NASA Task Load Index (NASA-TLX) survey is frequently used to capture developers’ perceived mental workload during coding tasks[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=Method%3A%20We%20propose%20a%20controlled,TLX). Many studies ask participants to rate their cognitive load or difficulty after tasks, providing subjective but valuable insight into their effort. Surveys and interviews also reveal qualitative aspects of comprehension (e.g., where a participant felt confused or what strategies they used).

Increasingly, researchers complement self-reports with **objective psycho-physiological measurements**. A recent mapping study identified dozens of experiments measuring cognitive load in software engineering, with 55% of them using **electroencephalography (EEG)** to track brain activity[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=Prior%20work%20has%20explored%20various,comprehension%20tasks%20using%20fMRI%20and). EEG offers a direct indicator of cognitive effort (e.g., certain brainwave patterns correlate with working memory load). In some cases, multiple sensors are combined – eye trackers, galvanic skin response, heart rate – to triangulate the cognitive state more robustly[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=Prior%20work%20has%20explored%20various,comprehension%20tasks%20using%20fMRI%20and). For instance, Fritz et al. combined EEG, eye-tracking, and skin sensors to successfully predict code task difficulty with up to 84% accuracy[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=al.%C2%A0,memory%2C%20attention%2C%20and%20language%20processing), showing that multimodal data can objectively signal when a developer is struggling. **Eye-tracking** in particular has become a centerpiece method for programming research. By recording where and for how long a developer looks at code, eye-tracking reveals fine-grained details of comprehension: which parts of code draw attention, scan patterns, and moments of back-and-forth navigation. Tang et al.’s study used eye-tracking to compare attention distribution on AI-generated vs. human code, and linked metrics like fixation duration to cognitive load[nztang.com](https://www.nztang.com/assets/files/papers/tang_vlhcc24.pdf#:~:text=3,W2)[nztang.com](https://www.nztang.com/assets/files/papers/tang_vlhcc24.pdf#:~:text=match%20at%20L548%20Finally%2C%20they,CONCLUSION%20AND%20FUTURE%20WORK). Long fixations or repeated passes over a code snippet often indicate higher mental effort or confusion. Researchers also measure **task performance** – e.g., time to complete a task or answer a comprehension question, and accuracy on quiz-like tasks about code. Response time and error rates are classic proxies for how difficult code is to understand. In controlled trials, differences in completion time between with-AI and without-AI groups (such as the 55% speed-up reported by Peng et al. with Copilot[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=mixed%20but%20generally%20positive%20results,34%5D%20found%20that%20while)) are interpreted alongside cognitive load measures, to see if faster performance comes at a cognitive cost.

Notably, **neurophysiological methods** have expanded what we can observe. Beyond EEG, some studies have even employed functional MRI to watch the brain in action as programmers read code, as mentioned above[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=electroencephalography%20%28EEG%29%2C%20while%2036,memory%2C%20attention%2C%20and%20language%20processing). fMRI is less common (due to cost and complexity) but it provided striking evidence that programming activates high-order cognition (memory, language centers). Another emerging technique is pupillometry (measuring pupil dilation), which can reflect mental effort on short time scales. In sum, today’s research arsenal ranges from **qualitative** (think-aloud protocols, interviews capturing strategy and confusion in developers’ own words) to **quantitative** (timings, correctness, eye gaze statistics, EEG signals, biometric stress indicators). Often these are used together: for example, a study might record eye movements and EEG while a developer debugs code, then follow up with a survey and interview. This mixed-methods approach helps cross-validate findings – e.g., if a participant reports high mental effort, do we also see physiological signs of strain and slower task performance? Such triangulation is increasingly important in the age of AI assistance, where *apparent ease* (AI writes the code quickly) might mask *hidden effort* (the user mentally double-checking the AI). As Al Haque et al. note, relying solely on traditional metrics or self-reports can be misleading; thus multi-dimensional data (combining subjective and objective measures) is now used to capture the full picture of comprehension effort with AI tools[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=Despite%20growing%20evidence%20that%20AI,unreliable%20for%20assessing%20cognitive%20load).

**Readability and Surface-Level Code Features: Do They Still Matter?**

**Code readability** – the human-friendly quality of source code – is as crucial as ever, even with AI in the mix. The consensus in recent studies and industry surveys is clear: developers continue to highly value readable code in the age of LLMs[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=How%20and%20why%20code%20readability,perceived%20that%20the). In one industrial survey (118 practitioners at Atlassian), 81% of respondents affirmed that code readability remains important for their work, primarily because readable code *reduces long-term maintenance costs* and eases teamwork[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=How%20and%20why%20code%20readability,perceived%20that%20the). This sentiment holds despite the introduction of AI code generation; in fact, readability may be even *more* important when code is produced quickly by an assistant, since future engineers (or the users themselves) must be able to understand and safely modify that AI-produced code[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=At%20Atlassian%2C%20code%20readability%20has,its%20value%20in%20this%20LLM)[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=readability%20is%20amplified,for%20Atlassian%2C%20since%20it%20directly). Readability is not an academic abstraction but a practical concern: as one famous adage goes, *“code is read far more often than it is written.”* Empirical data backs this – developers spend up to 70% of their time reading code rather than writing it[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=research%C2%A0,significantly%20reduces%20software%20maintenance%20costs) – so any tool or practice that ignores readability risks hurting productivity downstream.

Crucially, classic surface-level features like **identifier names, comments, and code formatting** still have a profound impact on comprehension. A recent formal model of identifier naming quality demonstrated that *good names act as cognitive beacons*: they convey intent and trigger the right domain knowledge in the reader’s mind, thereby accelerating understanding. Conversely, **poor or ambiguous names** measurably increase cognitive load and hinder comprehension. For example, replacing a descriptive variable name with a cryptic abbreviation can *“dramatically [worsen] developers’ code comprehension performance,”* whereas clear, consistent naming conventions *improve comprehension speed and reduce errors*. These findings, drawn from large-scale studies (hundreds of thousands of identifiers and human ratings), reinforce that readable naming and styling are not mere preferences – they have real cognitive effects on developers. In practice, this means that even if an AI writes a function correctly, using meaningless placeholder names or convoluted structure can make it hard for a human to trust and maintain that code.

Research also shows that AI assistants can influence these surface aspects of code. Interestingly, when programmers use tools like Copilot, the resulting code tends to have **more predictable, conventional naming**. In a controlled study on Copilot’s impact on naming, identifiers chosen with AI suggestions had significantly lower entropy (i.e. were more predictable) than those chosen manually[advait.org](https://advait.org/files/lee_2024_copilot_predictability.pdf#:~:text=experiment%20,even). On one hand, this can be positive – predictable names often follow common conventions, which might improve baseline readability or consistency across a codebase. On the other hand, there is a trade-off: highly predictable names could be overly generic, sacrificing some descriptiveness in specific contexts[advait.org](https://advait.org/files/lee_2024_copilot_predictability.pdf#:~:text=straightforwardly%20predicted%20from%20prior%20code%2C,therefore%20focuses%20on%20this%20tradeoff)[advait.org](https://advait.org/files/lee_2024_copilot_predictability.pdf#:~:text=predictable%20identifiers%2C%20which%20may%20sometimes,initiative%20programming%20tools%2C%20and). Developers face an *“attention investment”* choice here: accept the AI’s quick suggestion (usually a generic name) or invest extra thought to refine the name for clarity[advait.org](https://advait.org/files/lee_2024_copilot_predictability.pdf#:~:text=Mixed,attention%20investment%20model%20are%20both). The fact that this choice exists confirms that surface-level quality is still on developers’ minds. In practice, many developers mitigate AI’s tendencies by actively reviewing and renaming identifiers or adding comments for clarity – essentially performing the human oversight needed to keep code readable.

Encouragingly, evidence so far suggests that **AI-generated code can be as readable as human-written code** when guidelines are followed. In the Atlassian study, the team compared 144 instances of LLM-generated code to human code in their enterprise setting and found readability metrics to be *comparable*, with only negligible differences in line length, complexity, and comment density[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=How%20readable%20is%20the%20LLM,generated%20code). A majority of practitioners in that survey even perceived AI-generated code to be **more or equally readable** as human code (39% said more readable, 34% said about the same)[arxiv.org](https://arxiv.org/html/2501.11264#:~:text=How%20and%20why%20code%20readability,perceived%20that%20the). These perceptions likely stem from the fact that modern LLMs have been trained on well-structured code and often produce idiomatic patterns. However, this comparability holds when AI is guided properly – for example, by prompt instructions or coding standards. If an AI is prompted to write *clean, “easy-to-read” code*, it tends to do so. A recent experiment directly demonstrated this: when researchers prompted an AI to generate *“easy-to-read”* code, the resulting Python snippets yielded significantly lower cognitive workload for developers in comprehension tasks compared to equivalent human-written code[elib.dlr.de](https://elib.dlr.de/204978/#:~:text=electrical%20brain%20activation%20with%20an,generated%20code). In that study, both objective measures (EEG brain activity) and subjective ratings showed that **readable AI-generated code reduced mental effort**, whereas AI code optimized only for efficiency (performance) was harder to read[elib.dlr.de](https://elib.dlr.de/204978/#:~:text=electrical%20brain%20activation%20with%20an,generated%20code). This finding highlights that **readability still matters in practice** – so much that we can measure its effects on the brain – and that we can actively influence it through tooling and conventions. In conclusion, far from diminishing the importance of readable code, the advent of AI assistants has **amplified** it: to safely harness AI-generated code, developers rely on clear naming, documentation, and simple designs to comprehend what the AI has done. Readability remains a cornerstone of program comprehension and an active area of research in this new context.

**Proposed Thesis Direction: *Cognitively Grounded Readability in Human–AI Programming***

Building on the current readability model (particularly the four-factor identifier readability framework), a promising direction for a thesis is to **extend code readability research with explicit cognitive load modeling in the context of AI-assisted development**. The core idea is to develop a *human-focused* model of code readability that not only considers static code features (like naming, style, complexity) but also incorporates how these features impact a developer’s cognitive load and comprehension process, especially when interacting with AI suggestions. This extension responds to the emerging evidence that certain code qualities (e.g. clear identifiers, consistent style) directly correlate with reduced mental effort[elib.dlr.de](https://elib.dlr.de/204978/#:~:text=electrical%20brain%20activation%20with%20an,generated%20code). It also addresses new questions: for example, *how do AI interventions (auto-completions, code rewrites) affect the distribution of cognitive effort in a coding task?* and *where are the comprehension bottlenecks when humans and AI collaborate on code?* The thesis will remain **human-centric**, focusing on developers’ cognitive processes and bottlenecks (e.g. moments of high cognitive load, confusions, or misinterpretations), with the goal of improving both our theoretical understanding and practical ability to alleviate those issues.

**Thesis Objectives and Scope**

* **Integrate Cognitive Load Measures into Readability Models:** We aim to augment existing readability metrics with a quantitative cognitive dimension. For example, the current identifier readability model could be expanded by weighting its factors (Meaningful Clarity, Naming Conformance, etc.) according to their impact on actual cognitive load (as measured by experiments). This will result in a mathematical model that predicts *comprehension effort* for a given code snippet. The model might take the form of a regression or probabilistic model where input features include classic readability metrics and perhaps new features capturing AI-related aspects (e.g. was the code AI-generated, does it contain patterns known to confuse developers). The output would be an estimated cognitive load score or probability of misunderstanding. We will formulate this model with statistical rigor – potentially using multiple linear regression or machine learning, trained on empirical data – ensuring it is **mathematically grounded** and interpretable.
* **Identify Comprehension Bottlenecks in AI-assisted Coding:** Using empirical studies, the thesis will pinpoint which scenarios or code characteristics cause spikes in cognitive load for developers. For instance, do certain AI-generated constructs (like very compact “efficient” code) consistently burden human working memory? Does mixing different naming styles (from human vs AI contributions) create confusion? We will employ **user studies** to observe developers performing code understanding tasks under various conditions: reading clean code vs. less readable code, using AI explanations vs. manual reasoning, etc. During these tasks we will collect data such as eye-tracking (to see what parts of code demand more attention), response times for comprehension questions, error rates in explaining code, and physiological signals (EEG or pupil dilation) for cognitive load. This empirical foundation ensures the model and thesis insights are grounded in reality. For example, if eye-tracker data shows that developers frequently reread a particular line or skip back-and-forth (high fixation count), that line is likely a comprehension bottleneck – we can then correlate that with code features (maybe a poorly named variable or a complex expression) to validate the model’s predictions.
* **Human-in-the-Loop Readability Enhancement:** As a forward-looking component, the thesis can explore how *AI itself might assist in improving readability* and reducing cognitive load, closing the loop between human and AI. For example, we could prototype an “AI pair programmer” that not only generates code but also suggests renaming identifiers or reformatting code for clarity based on the readability+cognitive load model. This involves mining datasets of code revisions: do pull requests that improve naming or add comments correspond to measurable drops in cognitive effort for future readers (perhaps approximated by fewer bug fixes or faster code reviews)? By analyzing large code corpora (e.g. mining thousands of code review comments that request clearer naming), we can validate that the surface-level improvements our model champions are actually practiced and valued in the real world. This mining complements the user studies, giving a **big-data empirical angle** alongside lab observations.

**Methodology**

The research will follow a mixed-method, multi-phase methodology:

1. **Literature and Model Synthesis:** We will start by reviewing existing readability models and cognitive theories (covering 2000–2025 literature) to synthesize a comprehensive framework. This yields initial hypotheses (e.g., *“meaningful naming reduces cognitive load”*; *“AI suggestions might introduce comprehension overhead unless they adhere to human-oriented standards”*). We will formalize an *extended readability model* that incorporates these hypotheses (for instance, adding a “cognitive load penalty” for certain code patterns, or a factor for consistency when human and AI code intermingle).
2. **Experimental Design:** We will design controlled experiments to measure developers’ comprehension effort on code snippets that vary systematically in readability and origin. For example, one experiment might give participants two functionally identical code snippets – one written in a clean, self-explanatory style, another deliberately made less readable (poor names, no comments, or generated by AI under an “efficient” prompt). Participants will perform understanding tasks on each. During tasks, we collect (a) **eye-tracking data** (to capture attention and detect confusion points), (b) **EEG readings** or secondary-task reaction times (for objective cognitive load), and (c) **NASA-TLX surveys** for subjective workload. We might also utilize **think-aloud protocols** to qualitatively capture where participants struggle. Another set of studies could involve AI assistance: participants solve a problem with AI help in one condition and without in another, letting us observe differences in cognitive effort (similar to studies by Vaithilingam, Peng, etc., but focusing on *comprehension* of the resulting code). All experiments will be conducted with statistically sound sample sizes and within-subject designs where possible, to enable robust comparisons.
3. **Data Analysis and Model Refinement:** Using the collected data, we will quantitatively analyze which factors most strongly predict cognitive effort. This could involve regression analysis where the dependent variable is a cognitive load metric (e.g. peak pupil dilation, or EEG theta-band power, or simply time to comprehend) and independent variables are code readability scores (from our model) and other descriptors (code length, complexity metrics, whether code was AI-generated, etc.). We expect to validate that factors like *Meaningful Clarity* of identifiers have a significant inverse correlation with cognitive load, for example, and we will quantify this relationship. If needed, we’ll refine the model – e.g., adjusting weightings of factors – to best fit the empirical data (similar to how readability models are tuned to match human judgements). The outcome will be a **mathematically defined model** (potentially a formula or algorithm) that can take a piece of code and estimate the cognitive effort required to comprehend it. We will also visualize results with diagrams: for instance, a graph might show how comprehension time increases as identifier clarity (as scored by our model) decreases, illustrating the model’s predictive power.
4. **Thesis Validation and Case Studies:** To demonstrate the model’s usefulness and generality, we will apply it in several contexts. One validation might use an independent dataset: e.g., take a set of code examples from open-source projects along with human comprehension difficulty ratings (or outcomes of comprehension quizzes), and show that our cognitive-readability model correlates strongly with those outcomes, outperforming traditional complexity metrics. We also plan a **case study in an AI-assisted setting**: integrate our model into an IDE plugin that analyzes code as it is being written (or generated by AI) and flags potential comprehension hurdles (like “this variable name is low clarity” or “this function is very long – cognitive load likely high”). By monitoring developers using this tool, we can gather feedback and see if it leads to more self-explanatory code being produced, thereby closing the empirical loop. Throughout, we will use diagrams and flowcharts to elucidate the framework (e.g., a flowchart of the experimental procedure, and an architecture diagram showing how code features feed into the cognitive load model and then into a tool).

**Expected Contributions and Novelty**

This thesis will contribute a novel *cognitively-grounded readability model* that extends prior work by explicitly linking code characteristics to human cognitive effort. Its contributions are poised to be: (1) a theoretical framework synthesizing readability and cognitive load (bridging a gap between software metrics and human cognitive science), (2) an empirically validated model (with strong statistical backing from both lab studies and real-world data) that can predict comprehension difficulty, and (3) practical guidance and tools for improving code readability in the presence of AI – for example, recommendations for AI code assistant design (to prioritize suggestions that minimize cognitive load for the user). The human-focused perspective ensures the work keeps developer *experience* and *well-being* central, recognizing that productivity gains are moot if they come at the cost of mental fatigue or misunderstanding[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=While%20some%20research%20shows%20significant,when%20collaborating%20with%20AI%20assistants)[arxiv.org](https://arxiv.org/html/2501.02684v1#:~:text=Despite%20growing%20evidence%20that%20AI,unreliable%20for%20assessing%20cognitive%20load). By quantitatively capturing comprehension bottlenecks, the thesis could, for instance, identify “cognitive hot spots” in programs (regions of code that consistently cause high mental effort) and propose mitigations (like refactoring patterns or tool support to alleviate those). This goes beyond existing readability models by not just saying *what* looks readable, but showing *how much* a given improvement might reduce a developer’s cognitive load.

Importantly, the work will be scoped to ensure it is a **publishable and novel research contribution**. While prior studies have tackled code readability and some have measured cognitive load, none has fully integrated these aspects into a predictive model for the age of AI-assisted coding – this thesis would be the first to do so. The inclusion of physiological measurements in evaluating code comprehensibility is still relatively novel in software engineering, providing a cutting-edge angle. Additionally, focusing on the *interaction* of humans with AI-generated code (an area only beginning to be explored) ensures the thesis breaks new ground. The empirical approach (e.g. combining eye-tracking with code analysis) will yield rich data that can lead to multiple publications – for example, one paper on the correlation of identifier quality with EEG signals, another on the differences in cognitive load for AI-written vs. human-written code. The work’s novelty also lies in its interdisciplinary nature: it applies theories and methods from psychology (cognitive load theory, human-computer interaction) to a pressing software engineering problem.

**Outline of the Thesis (Proposed)**

1. **Introduction** – Presents the research problem and motivation: the enduring importance of human program comprehension in an era of AI code generation. Outlines the thesis goals and contributions, with a motivating example (e.g., a scenario where an AI-generated snippet caused difficulty due to poor readability).
2. **Literature Review** – Reviews relevant work in program comprehension, code readability models, cognitive load in software engineering, and human-AI interaction in programming. This establishes the knowledge gap: while code readability and cognitive effort have each been studied, they have not been jointly addressed in the context of AI-assisted development.
3. **Background and Theoretical Framework** – Introduces the theoretical foundation, including cognitive models (working memory limits, attention investment model, etc.) and the current readability models (e.g., the four-factor identifier readability model). Develops the extended framework that hypothesizes links between readability factors and cognitive load (supported by initial evidence from prior studies). May include a conceptual diagram of how code features influence the developer’s cognitive process.
4. **Research Questions and Hypotheses** – Clearly states the research questions (for example: *RQ1: How do specific code readability factors affect developers’ cognitive load? RQ2: In what ways does AI-generated code alter the comprehension process? RQ3: Can we quantitatively predict comprehension difficulty from code attributes?*) and associated hypotheses (e.g., *H1: Code with higher identifier clarity will yield lower cognitive load measures (lower NASA-TLX scores, fewer fixations, etc.)*; *H2: Developers using AI-generated code with low readability will exhibit more frequent misunderstandings or slower debug times, compared to readable code*).
5. **Methodology** – Details the research design. Describes the study participants, tasks, and materials (code snippets, AI assistant setup, etc.). Explains the instruments and data collection techniques: how eye-tracking will be set up, what EEG metrics or survey scales will be used, how code readability will be scored (using or extending the existing model). Outlines the procedure for each experiment or data collection phase. Also describes any mining of repositories or industry data, if applicable. Ensures replicability by providing statistical plans (sample size rationale, analysis methods). A flowchart here might illustrate the experimental workflow, from presenting code to a participant through measuring their responses and analyzing results.
6. **Model Development** – Presents the construction of the new readability-cognitive load model. This section will likely include mathematical formalisms: for instance, the formula combining the four identifier factors into a single readability score, and then our extended formula adding a cognitive load term. It will report how we calibrated the model using the experimental data (e.g., regression coefficients, goodness-of-fit). Tables or graphs will show the relationships discovered (e.g., a table might list the beta weights of each factor in predicting cognitive effort, with p-values). If machine learning is used, this section would describe the features and the performance of the predictive model (with metrics like R² or mean error for predicting comprehension time).
7. **Results** – Documents the findings of the empirical studies in detail. This includes results for each hypothesis: for example, a subsection on “Effect of Identifier Quality on Cognitive Load” would report that clear naming yielded significantly lower average cognitive load scores than unclear naming (with statistical evidence). Another subsection might present differences between human-written vs AI-written code comprehension (perhaps showing, as in the DLR study, that *easy-to-read AI code* led to significantly less perceived effort than human code[elib.dlr.de](https://elib.dlr.de/204978/#:~:text=electrical%20brain%20activation%20with%20an,generated%20code)). We will also report any surprising findings, such as cases where a factor thought important had no effect on measured comprehension (null results), which could refine the theory. Visualizations (bar charts, line graphs of cognitive load over time, heatmaps of eye gaze on code, etc.) will be used extensively to make the results digestible.
8. **Discussion** – Interprets the results in the context of the research questions. It will explain how the findings support or refine our cognitive readability model. For instance, if we found a strong correlation between the “Domain Relevance” of names and reduced misunderstandings, we discuss why domain-specific cues likely aid memory. We also consider the implications for software practice: e.g., should coding standards emphasize certain readability aspects more strongly? Is there evidence that AI assistants should be tuned to produce more human-readable outputs by default? We will relate our results to existing literature – confirming, extending, or challenging prior assumptions. Importantly, this section will highlight the *human-centered implications*: how adopting a cognitive perspective on code quality can improve developer productivity and well-being (preventing fatigue, reducing errors). It may also include a running example revisited, showing how our model could have predicted or prevented a real-world comprehension issue.
9. **Thesis Conclusion and Future Work** – Summarizes the contributions, noting that we developed a novel model and provided empirical validation for the continued importance of human-oriented code comprehension in the AI era. It will enumerate the thesis’s key contributions (e.g., “first comprehensive model linking code readability to cognitive load”, “large-scale evidence that readable code mitigates cognitive bottlenecks”, etc.). This section also outlines future research avenues opened up by this work: for example, applying the model to educational settings (helping novices write more comprehensible code), or extending it to consider *team* comprehension (how readability factors play out in code reviews with AI-generated code). We might suggest exploring other cognitive measures (like fMRI or new biometric devices) or integrating the model with AI assistants so that the AI can optimize for readability. The thesis will close on the note that **program comprehension research is not only still relevant, but is enriched by interdisciplinary approaches**, ensuring that as software development advances with AI, human cognitive needs remain in focus.

By pursuing this thesis, we expect to deliver a publishable, novel contribution that solidifies the role of human cognition in modern software engineering. It will provide both theoretical insight – updating the conceptual models of program comprehension for the 2020s – and practical tools or guidelines to improve code readability and, by extension, developer efficiency and satisfaction in an AI-assisted world.