

Bachelorarbeit

Untersuchung von KI-Methoden zur Optimierung von Software

Fakultät für Physik
Lehrstuhl für Elementarteilchenphysik
Ludwig-Maximilians-Universität München

vorgelegt von
Philip Obi

7. August 2025



Vorgelegt zur Erlangung des Grades B.Sc.
Betreut von Prof. Dr. Thomas Kuhr

Bachelor's Thesis

Exploration of AI Methods for Software Optimization

Department of Physics
Chair of Elementary Particle Physics
Ludwig-Maximilians-Universität München

submitted by
Philip Obi

7 August 2025



Submitted for the degree of B.Sc.
Supervised by Prof. Dr. Thomas Kuhr

Abstract

This thesis explores the potential of artificial intelligence (AI) methods for software development, specifically code optimization tasks.

The research focuses on evaluating these capabilities within the complex codebase of Belle II's basf2 software, testing performance in code analysis, documentation, and optimization during the refactoring of basf2's V0Fitter. Performance was also assessed on generic C++ code containing common inefficiencies.

Four large language models (LLMs) were evaluated: Claude Sonnet 4, Gemini 2.5 Pro, GPT-4o, and o4-mini, with Claude Sonnet 4 demonstrating the highest performance overall.

The refactored V0Fitter achieved over 50% reduction in per-event execution time compared to the current implementation. Testing revealed that current LLMs face significant limitations when autonomously optimizing code in complex software environments such as basf2, often struggling with intricate logic and producing inconsistent results. Nevertheless, they proved valuable for code analysis, codebase exploration, and bug detection.

For medium- to low-complexity tasks, LLMs demonstrated improved autonomous optimization performance, though code quality varied by model and specific example.

Beyond formal testing, AI tools significantly accelerated development through natural language querying and contextual understanding.

However, notable concerns include potential for confident incorrect responses, code vulnerabilities, and environmental impact considerations.

Acknowledgments

I would like to thank Prof. Dr. Thomas Kuhr for providing me with the opportunity to conduct my bachelor's thesis in his research group.

My special thanks go to my supervisors Dr. Thomas Lück and Dr. Nikolai Krug for their invaluable support, insightful guidance, and continuous mentorship throughout the entire project.

I would also like to thank my family and friends for their constant support, encouragement, and patience, which have been a great help throughout my studies and especially during the completion of this thesis.

Contents

1	Introduction	1
1.1	The Belle II Experiment	1
1.2	The Belle II Detector	2
1.3	The Belle II Analysis Software Framework	3
1.3.1	Implementation Language	4
1.3.2	Code Organization and Architecture	4
1.3.3	Detector Data Processing Flow	5
1.4	Artificial Intelligence: Relevant Technologies and Concepts	6
1.4.1	Historical Context and Evolution	6
1.4.2	Large Language Models	6
1.4.3	Reasoning Models	7
1.4.4	Retrieval-Augmented Generation	8
1.4.5	Agentic AI	8
1.4.6	Real-World Application: GitHub Copilot	9
1.4.7	Visual Studio Code and the GitHub Copilot Extension	9
2	V0Fitter Optimization	11
2.1	Introduction	11
2.2	Initial Code Analysis and Profiling	13
2.3	In-depth Code Analysis and Refactoring	16
2.3.1	Short background Information and Class Explanations	17
2.3.2	Overview of Performed Optimizations	18
2.3.3	Overview of Refactored Code Architecture	19
2.4	Evaluating Computational and Reconstruction Performance of the V0Fitter Implementations — Methodology and Results	20
2.4.1	Data Generation, Validation and Profiling Steering Scripts	20
2.4.2	Testing Methodology	23
2.4.3	Comprehensive Testing Results for the V0Fitter Implementations	24
2.5	Documentation of AI Usage — Methods and Experiences	25
2.5.1	Refactoring advice	26
2.5.2	Documentation Generation for the V0Fitter Refactor	27
2.5.3	V0Fitter Optimization Test	28
2.5.4	Use of AI during coding	32
2.6	Additional Tools and Methods	34
2.6.1	OverlayFS	34
2.6.2	VS Code GDB Debugging	34
2.6.3	V0Fitter debugging	35
2.6.4	Clangd Extension	35
2.6.5	Chat Exporting	36
3	Generic Code Optimization	37
3.1	Methodology	37
3.2	Results	38
4	Model Selection	44
4.1	Selection of Models for Comparison	44
4.2	AI Benchmarking	46
5	Chat Archiver Design and Implementation	47
6	Summary	49
7	Outlook	50
8	Appendix	55

Note on Repository Structure and Code References

I have organized all code and supplementary materials developed during this thesis into topic-specific repositories, including the LaTeX source code for this document itself.

Throughout this work, I reference these repositories in footnotes to provide access to exact implementations, additional documentation, and extended reading materials.

For brevity, repository links are displayed in the shortened format [github/philipobi-bachelor/<repository path>](https://github.com/philipobi-bachelor/<repository path>), which corresponds to the full URL <https://github.com/philipobi-bachelor/<repository path>>.

In digital versions of this thesis, these links are clickable and direct readers to the specific resources. Additionally, a unified repository at <https://gitlab.physik.uni-muenchen.de/agkuhr-bachelors-2025/philipobi-ai-software-optimization> contains all relevant repositories as submodules.

Clicking on any top-level element in this unified repository automatically redirects to the corresponding specific repository.

1 Introduction

The increasing complexity of modern particle physics experiments presents unprecedented challenges for software optimization and analysis.

Contemporary high-energy physics experiments generate vast amounts of data, requiring sophisticated software frameworks to process, reconstruct, and analyze particle interactions. As these systems grow in complexity — incorporating millions of lines of code, numerous interdependent modules, and intricate data processing pipelines — traditional optimization approaches often struggle to identify effective performance improvements. This complexity motivates the exploration of AI methods to enhance understanding, development, and optimization processes within large software systems, particularly those used in physics applications.

The Belle II experiment’s software framework, basf2 [1][2], exemplifies this challenge. Basf2 must process data from billions of particle collisions while maintaining high precision and efficiency. The project’s codebase is extremely large and complex, featuring code written in multiple programming languages, components inherited from different projects, and contributions from numerous authors (cf. Tab. 1). Additionally, documentation and coding conventions are not always consistent across the framework. If AI-driven optimization techniques prove effective in this context, they could provide a valuable pathway for improving computational efficiency and enhancing the overall scientific output of the experiment.

Language	Files	Lines		
		Blank	Comment	Code
C++	3 230	123 480	100 246	590 457
Fortran 77	464	70 023	139 064	515 100
Python	2 173	58 837	87 192	235 594
C/C++ Header	4 066	87 135	176 125	221 988
...				
Sum:	10 855	362 940	544 716	1 917 380

Table 1: Source code file distribution in the basf2 repository, grouped by programming language and sorted by lines of code (generated with cloc [3])¹

1.1 The Belle II Experiment [4]

The Belle II experiment is situated at the SuperKEKB electron-positron collider in Tsukuba, Japan. SuperKEKB is a next-generation B-factory, a particle collider specifically designed to produce large numbers of B mesons — composite particles consisting of a bottom quark and a lighter quark. Studying these mesons is essential for testing the predictions of the Standard Model (SM) and for probing potential physics beyond it.

SuperKEKB collides electrons and positrons at center-of-mass energies tuned primarily to the $\Upsilon(4S)$ resonance, a short-lived bound state of a bottom quark and its antiquark. With a mass of approximately 10.58 GeV, the $\Upsilon(4S)$ decays almost exclusively into particle-antiparticle pairs of neutral or charged B mesons. This makes it an ideal source for studying B meson properties in a clean and well-controlled experimental environment.

The central scientific objective of Belle II is to investigate the flavor sector of the Standard Model, i.e. interactions that distinguish between quark and lepton types (“flavors”), with particular emphasis on heavy flavor physics involving particles that contain bottom or charm quarks. These studies enable precision tests of the SM and offer sensitivity to indirect effects of new, undiscovered particles or interactions.

Of particular importance in heavy flavor physics is CP violation, an asymmetry in the behavior of particles and their antiparticles under the combined transformations of charge conjugation (C) and parity (P). Investigations of CP violation are essential to understanding the matter-antimatter asymmetry of the universe, as this phenomenon is predicted by the SM, but not at the level observed in cosmological data. This gap between theory and observation makes it a compelling area for discovering new physics.

To achieve its scientific goals, Belle II aims to collect an integrated luminosity² of 50 ab^{-1} , representing a fifty-fold increase over the data collected by its predecessor, the Belle experiment. This massive dataset significantly

¹For the complete table see github.com/philipobi-bachelor/project/artifacts/bASF2-LOC

²In particle physics, (instantaneous) luminosity \mathcal{L} relates the interaction rate dR/dt to the production cross-section σ_p (measure of interaction probability): $dR/dt = \mathcal{L} \sigma_p$. While \mathcal{L} determines the interaction rate, the integrated luminosity $\int \mathcal{L} dt$ is a measure of the total data collected, since it is proportional to the number of events observed. [5]

enhances sensitivity to rare decays and subtle effects in the physics of B, D, and τ particles.

SuperKEKB operates with asymmetric beam energies (e^- : 7 GeV, e^+ : 4 GeV), resulting in a boost of the center-of-mass system, which causes produced B mesons to travel a measurable distance in the laboratory frame before decaying. The longer decay path allows for a more accurate measurement of the decay-time difference between produced pairs of a B meson and its antiparticle, a quantity essential for studying CP asymmetries.

The combination of high luminosity, clean experimental conditions, and advanced detection capabilities establishes Belle II as a leading facility for precision measurements in flavor physics and the search for physics beyond the Standard Model.

1.2 The Belle II Detector [4][6]

The Belle II detector employs a layered, cylindrical design where different subsystems work together to provide comprehensive particle detection and measurement capabilities. Each component is optimized to handle the challenging environment of high luminosity e^+e^- collisions while maintaining high resolution and efficiency across a wide range of particle momenta.

The following paragraphs provide an overview of the most important detector components and their functions, starting from the innermost layers and moving outward. For a schematic representation of the Belle II detector and its main components, see Figure 1.

Interaction Point and Beam Pipe The electron and positron beams collide at the interaction point (IP), located at the center of the Belle II detector. By convention, particle positions and momenta are described in a coordinate system where the z-axis is aligned with the solenoid’s central axis and points in the direction of the electron beam, and the origin lies in the interaction point (Fig. 1, right).

The beam pipe, which also coincides with the z-axis, contains the colliding beams and is maintained under high vacuum conditions to minimize beam-gas interactions that produce unwanted background events. Near the interaction point, it is constructed from beryllium, with a cylindrical inner radius of 10 mm. The element beryllium was chosen for its favorable properties: it provides adequate vacuum containment while maintaining minimal material thickness. This design consideration is important because particles produced in collisions must traverse the beam pipe material, and excessive thickness would degrade the precision of vertex and momentum measurements. When particles pass through the material, they experience energy loss through ionization and multiple scattering processes, effects that directly impact particle reconstruction accuracy and must be properly accounted for in both the tracking algorithms and subsequent physics analyses.

Vertex Detector System The vertex detector system (VXD) consists of two layers of pixel detectors (PXD) and four layers of silicon strip detectors (SVD) arranged around the beam pipe. The pixel detectors form the innermost layers and provide precise position measurements, while the strip detectors constitute the outer layers. When charged particles traverse these silicon sensors, they deposit energy that generates electrical signals. This system measures particle trajectories close to the interaction point, enabling accurate reconstruction of particle production and decay vertices.

Central Drift Chamber The Central Drift Chamber (CDC) is a cylindrical gas-filled detector that surrounds the VXD. When charged particles traverse the gas volume, they ionize atoms along their trajectory. The freed electrons subsequently drift toward sense wires under an applied electric field, creating electron avalanches that produce measurable signals upon arrival. Using geometric triangulation, particles’ positions at the time of initial ionization can be reconstructed from the measured signal times of individual wires and the known electron drift velocity. This enables precise reconstruction of charged particle trajectories, from which particle momenta can be determined via their curvature in the magnetic field.

Particle Identification Systems Belle II employs two distinct Cherenkov detectors for particle identification in different detector regions. They both exploit the fact that charged particles with identical momentum but different masses travel at different velocities, producing distinct Cherenkov light signatures that allow reliable particle type determination.

The barrel region, which covers the central cylindrical area around the interaction point, houses the Time-of-Propagation (TOP) counter. This detector records both the positions and the arrival times of Cherenkov photons produced by charged particles traversing quartz radiator bars. Different particle species produce Cherenkov

photons at characteristic angles depending on their velocity, resulting in distinct patterns of photon arrival times and positions at the detector.

The forward endcap region, covering the area along the beam direction where particles are boosted due to the asymmetric beam energies, contains the Aerogel Ring Imaging Cherenkov Detector (ARICH). This system uses aerogel as the radiating medium and detects the characteristic ring patterns formed by Cherenkov photons, with the ring radii depending on the particles' velocities.

Both detectors achieve particle identification by combining their respective Cherenkov signatures with momentum measurements from the CDC, enabling mass determination through the relationship between momentum, velocity, and mass.

Electromagnetic Calorimeter The electromagnetic calorimeter (ECL) is made up of thousands of thallium-doped cesium iodide crystals arranged in layers surrounding the interaction region. When high-energy photons or electrons enter the crystals, they initiate electromagnetic showers through repeated bremsstrahlung and pair production processes. These showers generate scintillation light within the crystals, with the intensity of the emitted light being proportional to the energy deposited by the particle, enabling the measurement of incident particles' energies.

Muon and K-long Detection System The outermost detector component is the K-long and muon detection system (KLM), which consists of alternating iron plates and active detector elements. K-long particles (neutral kaons) interact hadronically within the iron absorbers and produce detectable particle showers. Muons are identified primarily by their ability to penetrate through all inner detector layers and the iron absorber with minimal interaction, unlike most other charged particles. As they pass through the active KLM elements, they deposit energy via ionization, allowing them to be traced through the entire detector, and enabling their identification.

Identification of muons is crucial for many important B-meson decays, and the KLM also facilitates the detection of neutral kaons that would otherwise remain undetected.

Superconducting Solenoid A superconducting magnet surrounds most of the detector components, creating a uniform magnetic field throughout the tracking region. This magnetic field causes charged particles to follow curved trajectories, with the radius of curvature directly related to the particle's momentum and charge. The magnetic field is essential for precise momentum measurements and charge identification of particles. The iron return yoke that contains and shapes the magnetic field additionally serves as absorber material for the KLM.

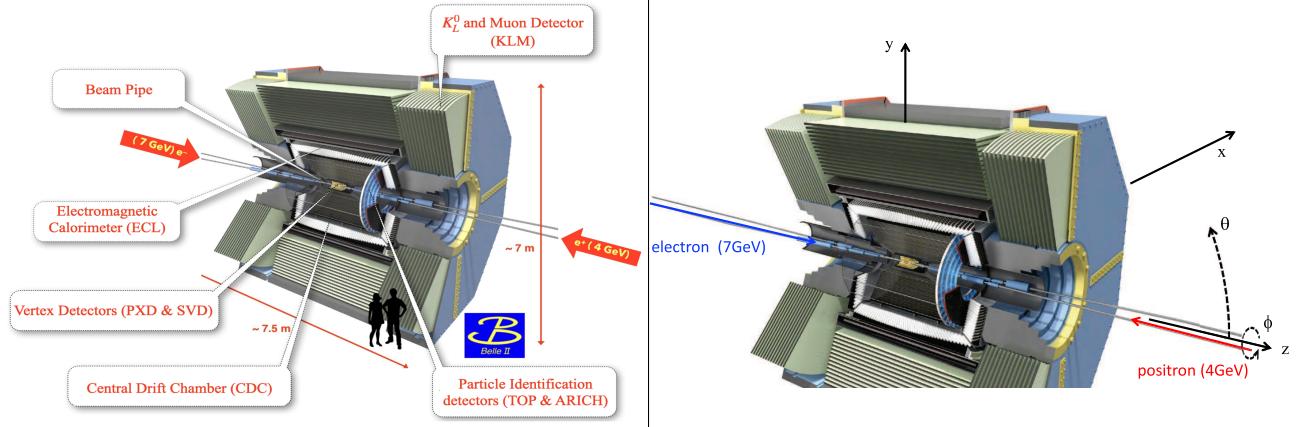


Figure 1:

Left: Cross-sectional view of the Belle II detector with labeled subsystems and beam configuration [7]

Right: Belle II experiment coordinate system (convention used in the Belle II software) [8]

1.3 The Belle II Analysis Software Framework [2][9][10]

The Belle II Analysis Software Framework (basf2) serves as the comprehensive software infrastructure for all data processing and analysis tasks within the Belle II experiment. This framework handles the complete data processing chain, from the initial unpacking of raw detector data to high-level physics analysis, including

simulation generation, event reconstruction and particle identification. Basf2 is the successor to basf, the analysis software for the first Belle detector.

Basf2's architecture aims to ensure three principles: modularity, flexibility and performance. Modularity is achieved through a component-based architecture where individual processing tasks are encapsulated in discrete modules that can be combined and reconfigured as needed. This in turn enables the flexibility to support diverse analysis workflows, from detector calibration to complex physics measurements, utilizing configurable processing chains. Performance considerations influence the choice of implementation languages and the framework's memory management strategies, ensuring efficient processing of the large volumes of data characteristic of modern high-energy physics experiments.

1.3.1 Implementation Language

The core data processing components, including all computationally intensive modules for reconstruction algorithms, are implemented in C++ (cf. Tab. 1). C++ was chosen due to the performance requirements of processing millions of collision events, where execution speed and memory efficiency are of critical importance, especially in the case of the trigger systems (see Sec. 1.3.3).

Python serves as the high-level interface and configuration language for basf2. All user interactions with the framework, such as the creation of analysis workflows and module configurations, occur through Python scripts, commonly referred to as steering files.

This allows to harness both the computational performance of C++ and the user-friendly syntax of Python

1.3.2 Code Organization and Architecture

The basf2 software stack distinguishes between external dependencies and core components developed specifically for Belle II. External components include third-party libraries and tools that provide foundational functionality, such as ROOT [11] for data analysis and visualization or Geant4 [12] for detector simulation. These externals are maintained in a separate repository from the core basf2 code.

The basf2 core includes the Belle-II-specific software, implementing data structures and processing routines for the individual detector components and more general reconstruction and analysis steps. The following paragraphs further explain the architecture of the basf2 core.

Packages A package in basf2 represents a logical collection of related functionality, typically organized around a specific detector subsystem or analysis tasks. Each package corresponds to a top-level directory in the basf2 repository and contains modules, utility functions and documentation related to its domain.

Modules A basf2 module represents a self-contained unit of code, usually implemented in C++, that performs a specific task within the data processing workflow. Each module implements a standardized interface that defines five main lifecycle methods, governing its behavior during execution:

- **initialize**: invoked once before event processing begins to perform any necessary initialization such as setting up resources and allocating memory
- **beginRun** and **endRun**: called at the start and end of each data-taking run³, allowing modules to perform run-dependent initialization or finalization tasks, such as resetting run-specific monitoring histograms
- **event**: contains the main processing logic and is called once for each processed event, implementing the core functionality of the module
- **terminate**: called once after all events have been processed and typically handles cleanup tasks or final calculations

Paths A path in basf2 defines an ordered sequence of modules that will process each event in turn. Conceptually, a path acts as a pipeline: each event is passed through the modules in the specified order, with each module performing its designated task (see Fig. 2 for a visualization). The path represents the complete workflow for a particular analysis or processing task, from initial data input to final output generation. Paths are constructed programmatically in Python steering files by adding modules in the desired execution order, allowing users to create custom processing chains.

³Run: a sequence of events with consistent detector conditions

The path-based approach provides flexibility in creating data processing workflows, enabling complex processing pipelines with support for parallel processing and conditional path execution. For conditional path execution, specialized modules can modify the processing path based on individual event characteristics (cf. path branching in Fig. 2, Left).

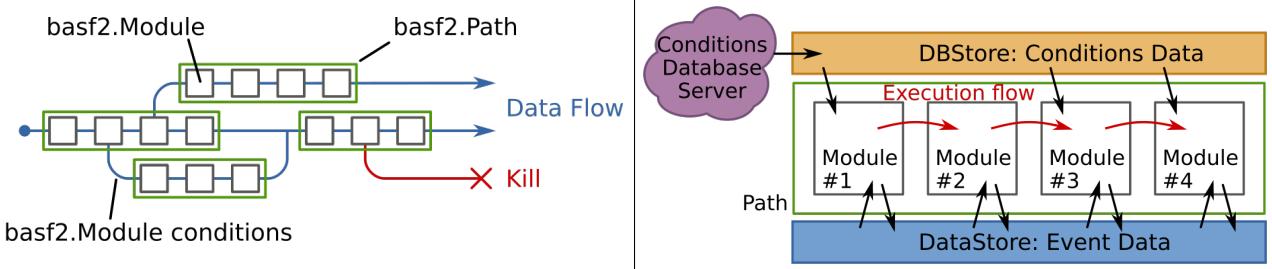


Figure 2: Schematic view of the processing flow in basf2 [13]

DataStore and StoreArrays The DataStore serves as a globally accessible interface for sharing data objects between modules. Instead of passing data directly between modules, all modules read from and write to the DataStore (see Fig. 2, Right). This is a key factor in maintaining the modularity of basf2’s processing and avoids tight coupling between individual modules in a path.

StoreArrays represent collections of objects of the same type within the DataStore. The framework provides functionality to model relationships between objects in different StoreArrays, such as the relationships between a track and the detector hits that it consists of. This allows for efficient navigation between different levels of the reconstruction hierarchy.

1.3.3 Detector Data Processing Flow

Trigger System The necessity for event triggering, i.e. a pre-selection of observed events before they enter the processing and storage pipeline, stems from both technical and physics considerations. From a technical perspective, the complete readout and storage of every collision event would overwhelm data acquisition systems and storage infrastructure. From a physics standpoint, most collisions produce well-understood background processes that, while scientifically valid, are not relevant for the specific research goals of the Belle II experiment. The trigger system enables filtering of interesting processes, such as B meson decays, that constitute the primary scientific targets of the experiment.

The trigger system operates in multiple stages to manage the high data rates from SuperKEKB collisions. The Level 1 (L1) trigger uses dedicated hardware to perform a (near) real-time selection of events based on reduced-resolution data from the CDC, ECL and KLM subsystems. The downstream High Level Trigger (HLT) on the other hand performs full event reconstructions using basf2, allowing for complex physics-based selection criteria such as invariant mass requirements and vertex quality cuts. The HLT’s reliance on basf2 for event processing places high performance requirements on the framework’s reconstruction algorithms, as any computational inefficiency directly impacts the experiment’s data acquisition capabilities and overall trigger acceptance rates.

Reconstruction Pipeline The reconstruction process to transform raw detector signals into physics objects suitable for analysis includes the following main steps: First, clustering algorithms group together signals from each detector subsystem that were likely produced by a common particle. Track finding algorithms then use this information to identify particle trajectories through the central detector systems, combining hits from the PXD, SVD, and CDC to create a complete picture of each particle’s path through the detector volume. Subsequent track fitting procedures reconstruct the kinematic variables along these trajectories as precisely as possible. The reconstructed tracks are then combined with additional data, such as energy deposits and timing information, to assign likelihoods for each track belonging to a certain particle species. Finally, during high-level analysis, composite objects are created by combining reconstructed tracks and clusters according to physics hypotheses.

1.4 Artificial Intelligence: Relevant Technologies and Concepts

1.4.1 Historical Context and Evolution

As outlined in the introductory section of the textbook “Artificial Intelligence: A Modern Approach” [14] — widely regarded as the standard text in the field — the development of artificial intelligence has been shaped by a series of major shifts in approach and methodology. In its early days, AI research was dominated by symbolic methods, which sought to represent knowledge and reasoning through explicit symbols and logical rules. These approaches aimed to model intelligent behavior by mimicking human reasoning processes.

Over time, researchers recognized the limitations of purely symbolic systems, especially when faced with the uncertainty and complexity of real-world environments. This realization led to the adoption of statistical and probabilistic methods, which allowed AI systems to learn from data and make decisions under uncertainty. The rise of machine learning marked a significant turning point, enabling computers to improve their performance through experience rather than relying solely on hand-crafted rules.

Another transformative phase in AI’s evolution came with the development and resurgence of neural networks. Inspired by the structure of the human brain, neural networks provided a new way to process information and recognize patterns.

Advances in computational power and learning algorithms eventually led to the emergence of deep learning, which has driven remarkable progress in areas such as natural language processing and computer vision.⁴

Today, AI continues to evolve rapidly, integrating ideas from its symbolic, statistical, and neural network roots.

1.4.2 Large Language Models (LLMs) [14]

LLMs represent the culmination of several decades of neural-network and deep-learning research. To understand how they arose, it helps to contrast them with earlier sequence models, such as simple recurrent neural networks (RNNs). RNNs process one token at a time and maintain a hidden state that is passed forward in time — a mechanism known as recurrence. While they can in principle capture dependencies across a sequence, in practice they struggle with very long contexts and suffer from numerical underflow or overflow during training.

Tokens and Embeddings LLMs operate on discrete units called tokens, which may be whole words, subwords, or characters, depending on the tokenization scheme. Each token is converted into an embedding: a dense vector of floating-point numbers that encodes semantic and syntactic information. Embeddings serve as the model’s “language of thought”, allowing text symbols to be represented in a numerical form suitable for neural network architectures.

The Transformer Architecture The transformer [16] replaces recurrence with a self-attention mechanism that allows every token to directly attend to every other token in a sequence. Intuitively, self-attention computes a weighted sum of all token embeddings, where the weights reflect context-dependent relevance. Transformers integrate multiple such attention blocks with simpler neural networks and stabilization mechanisms, enabling efficient, parallel modeling of long-range patterns.

Context and Context Size [17] In LLMs, the context refers to the sequence of input tokens over which self-attention operates. The context size (or context window) defines the maximum number of tokens the model can process simultaneously. This size is typically determined during training and remains fixed for the model. Context size is constrained by quadratic scaling of self-attention’s computational and memory requirements. Larger contexts enable longer-range dependencies and complex tasks like document summarization, but increase training and inference⁵ costs.

Pretraining, GPTs, and Fine-Tuning Modern LLMs are often built on the GPT (Generative Pretrained Transformer) architecture [18], which is a decoder-only transformer design. This means they consist solely of the decoder stack from the original encoder-decoder design and therefore employ causal self-attention, ensuring each position attends only to earlier tokens. During pretraining, GPT models learn to predict the next token

⁴For comprehensive coverage of machine learning, specifically neural networks and deep learning architectures, see also “Deep Learning” by Goodfellow et al. [15], a foundational textbook on these topics

⁵Inference: The stage in which a trained model processes new inputs, i.e. its application after training

in a text sequence. In this process, they acquire broad linguistic patterns from large-scale datasets, storing this knowledge in their parameters, the billions of numerical values that determine how inputs are transformed into generated outputs.

After pretraining, the resulting GPT foundation model can be fine-tuned on smaller, domain-specific datasets. This fine-tuning may involve updating all parameters or only specific layers, enabling the model to perform specialized tasks such as question answering or code generation.

Prompts and Few-/Zero-Shot Learning [19] A prompt is the natural-language text input provided to a LLM to specify a task or elicit a response. A notable feature of GPT-style LLMs is their ability to follow prompts with little or no additional training. In zero-shot mode, one supplies only an instruction, e.g. “Translate to French: ...”, and the model generates an answer directly. In few-shot mode, several example input-output pairs are prepended to the prompt, allowing the model to infer the desired behavior. These capabilities arise from extensive pretraining and the model’s generalization of learned patterns to novel tasks.

1.4.3 Reasoning Models [20]

Building on the foundations of pretraining, few-shot learning, and fine-tuning, a new class of models, often termed *Reasoning Models*, has emerged to address fundamental limitations of standard LLMs. Standard large language models face several critical challenges when attempting reasoning tasks. They often hallucinate plausible but incorrect answers, struggle to decompose complex problems into manageable sub-problems, and exhibit brittle performance when reasoning chains exceed a few steps. Additionally, their predictions can be overly influenced by the frequency of certain terms in training data rather than logical reasoning processes, suggesting reliance on memorized patterns rather than true inference capabilities.

Reasoning models address these limitations through three key techniques:

Chain-of-Thought (CoT) Prompting By prompting models to “think step by step”, CoT elicits explicit intermediate rationales before producing final answers. This transforms single-shot predictions into sequential reasoning processes, significantly improving performance on arithmetic, logic, and commonsense benchmarks. Several variants have emerged: Zero-shot CoT uses simple prompts like “Let’s think step by step” without examples, while rationale engineering techniques systematically improve the quality and effectiveness of reasoning demonstrations by optimizing how examples are constructed and selected.

Test-Time Scaling Unlike traditional scaling that increases model parameters, reasoning models dynamically allocate computational effort during inference (test time). Models learn to generate hundreds or thousands of reasoning tokens at test time, exploring multiple solution paths and self-correcting errors. This paradigm enables performance improvements through extended deliberation rather than larger model size.

Self-Correction and Verification Advanced reasoning models integrate self-consistency mechanisms that sample diverse reasoning chains and vote on consistent answers, alongside reflection processes that enable models to revisit and revise earlier reasoning steps when contradictions arise.

DeepSeek-R1 [21] is a reasoning model developed by the Chinese AI company DeepSeek, exemplifying these principles through a multi-stage training approach. The process begins with reinforcement learning applied to a base model (DeepSeek-R1-Zero) to develop reasoning behaviors. During this stage, the model learns through reward signals based on correctness, rather than supervised examples. This is followed by supervised fine-tuning with curated CoT data and iterative refinement through rejection sampling, a technique that generates multiple candidate responses and retains only the correct ones for training. This pipeline produces models capable of accurate and interpretable reasoning.

These developments represent a shift from next-token prediction toward incentivized inference, where models are explicitly trained to deliberate, verify, and self-correct rather than merely mimic text patterns. However, current reasoning models still face significant limitations, including struggles with complex real-world reasoning tasks, sensitivity to prompt formulation, and the ongoing question of whether they perform genuine logical inference or sophisticated pattern matching. Despite these challenges, reasoning models mark an important step toward more capable AI systems for complex scientific and engineering applications.

1.4.4 Retrieval-Augmented Generation [22]

Retrieval-Augmented Generation (RAG) addresses fundamental limitations of LLMs by combining their internal knowledge with external document collections for knowledge-intensive tasks. Traditional language models store information in their parameters during training, making it difficult to update their knowledge or provide sources for their outputs. RAG overcomes these challenges by retrieving relevant documents from an external corpus and using them as additional context during text generation.

The RAG approach consists of two main components: a retrieval system and a generation model. Before any queries are processed, documents in the external corpus are indexed by converting them into vector embeddings and storing them in a searchable database. During retrieval, the input query is similarly converted into an embedding and matched against the indexed documents to identify the most relevant passages. The generation model then produces output based on both the original input and the retrieved context, combining parametric knowledge from training with non-parametric knowledge from the external corpus.

RAG offers two operational modes: one that uses the same retrieved documents for generating the entire output, and another that can select different documents for each part of the output. This flexibility allows information synthesis from multiple sources when generating complex responses.

RAG demonstrates improved performance on knowledge-intensive tasks such as question answering, producing more factual outputs compared to standard language models. A key advantage is the ability to update knowledge by replacing the document collection without retraining the model. However, RAG introduces computational overhead and can be sensitive to retrieval quality, potentially producing errors when irrelevant information is retrieved.

1.4.5 Agentic AI [23]

Agentic AI systems represent a paradigm shift from reactive language model responses toward autonomous, goal-directed artificial intelligence. Unlike traditional LLMs that respond to individual prompts, agents exhibit goal-oriented behavior through sustained cycles of planning, execution, and adaptation.

Agent Architecture Components Modern agent architectures comprise several critical components enabling autonomous operation. Planning modules decompose complex objectives into manageable subtasks and sequence actions toward goal completion. Research identifies five major planning approaches: (1) task decomposition breaks problems into smaller sub-problems, (2) multi-plan selection evaluates multiple generated solution paths, (3) external module-aided planning leverages pre-existing external planners, (4) reflection and refinement revises plans based on new information, and (5) memory-augmented planning uses stored experiences to improve planning decisions.

Tool usage capabilities distinguish agents from standard language models by enabling interaction with external systems through function calling and API integration. Using tools, agents can access databases, execute code, send emails, retrieve web content, and manipulate external environments, expanding their problem-solving scope beyond text generation.

Memory systems provide both short-term working memory for immediate context and long-term storage for accumulated experiences.

Reflection and learning mechanisms enable self-improvement through systematic evaluation of past actions and outcomes, reducing errors and improving decision-making quality over time.

Multi-Agent Systems and Coordination Multi-agent architectures extend individual capabilities through collaborative problem-solving, organizing specialized agents into coordinated teams for parallel task execution. Two primary structures emerge: vertical architectures with leadership hierarchies and horizontal architectures emphasizing peer collaboration.

Effective systems employ structured communication protocols to prevent unproductive interactions and ensure relevant information reaches appropriate team members. Multi-agent systems have proven successful in scenarios requiring diverse perspectives or collaborative decision-making, though they introduce coordination overhead. Dynamic team formation, where agents are recruited based on task requirements, optimizes performance for specific problem phases.

1.4.6 Real-World Application: GitHub Copilot [24][25]

GitHub Copilot represents a prominent real-world implementation of the AI paradigms discussed previously, demonstrating how LLMs, retrieval-augmented generation, and agentic systems converge in practical software development tools.

Originally built upon OpenAI's Codex model [26], a specialized variant of GPT fine-tuned on publicly available code repositories, Copilot has evolved into a comprehensive AI-powered coding assistant developed by GitHub in collaboration with OpenAI.⁶

Hybrid Approach Integration GitHub Copilot embodies the hybrid AI approaches outlined in previous sections. Its RAG components manifest through code repository indexing, where the system maintains awareness of project structure, dependencies, and coding patterns specific to the current workspace. This contextual retrieval ensures suggestions better align with existing codebase conventions and architectural decisions.

Agentic features are evident in Copilot's ability to perform multistep tasks such as debugging workflows, where the system can analyze error messages, identify potential causes, and suggest comprehensive fixes. The chat interface supports complex problem-solving scenarios, allowing developers to engage in iterative refinement of solutions through natural language dialogue.

The integration of reasoning capabilities appears in Copilot's code logic verification and optimization suggestions, where the system can evaluate code quality, identify potential improvements, and explain the rationale behind its recommendations.

Performance and Limitations While GitHub Copilot has achieved widespread adoption⁷, it faces inherent limitations characteristic of current AI systems. Code quality varies significantly depending on task complexity, with simpler, well-established programming patterns receiving more reliable suggestions than novel or domain-specific implementations.

Security considerations remain important, as the system may inadvertently suggest code patterns that introduce vulnerabilities or expose sensitive information. Various safeguards have been implemented, including content filtering and bias detection mechanisms, but human oversight remains essential.

1.4.7 Visual Studio Code and the GitHub Copilot Extension

Visual Studio Code For source code editing tasks throughout this bachelor's thesis, I primarily used Microsoft's Visual Studio Code (VS Code) [28] due to its user-friendly interface and my prior familiarity with the environment. Additionally, it offers a wide range of useful extensions, such as the Remote Development Extension Pack [29], which allows connecting to remote VS Code servers via SSH, and accessing most of the functionality of the editor as if working on a local machine, even if operating systems differ between the client and host.

While GitHub Copilot is available in a limited form on GitHub itself, it is primarily used as an extension within integrated development environments (IDEs). GitHub Copilot is particularly well integrated into Visual Studio Code through dedicated extensions [30][31], which can be attributed to GitHub and VS Code both being Microsoft products.

As of late 2024, Copilot users can select from multiple language models, representing a significant change from earlier versions that used predetermined OpenAI GPT variants exclusively [32]. This flexibility allows developers to choose models based on their specific task requirements, with options ranging from fast completion-focused models to more sophisticated reasoning-oriented variants.

The VS Code extension implements GitHub Copilot through several key interfaces:

The Copilot Panel serves as the primary interaction hub and features a chat window that enables natural language communication with the AI system. This chat interface operates in three distinct modes: Ask mode for exploratory queries and code explanations, Edit mode for direct multi-file modifications, and Agent mode for autonomous task execution.

The panel supports extensive context integration through various mechanisms: Users can reference the entire codebase, target specific files, folders or symbol definitions, and utilize specialized tools such as web content

⁶OpenAI: US-based AI research company, backed by Microsoft but operates independently; pioneer in modern LLMs, introducing the GPT architecture [18]

GitHub: Platform for version control and collaborative software development; acquired by Microsoft in 2018.

⁷According to a recent press statement by Microsoft, more than 20 Million all-time users [27]

fetching capabilities (cf. Fig. 3, Right).

Code Completion Beyond the panel interface, the extension provides real-time code completion suggestions that appear as developers type. These suggestions range from single-line completions to entire function implementations based on contextual analysis of the surrounding code (see Fig. 3, Left).

Inline Chat Inline chat functionality extends this interactivity directly into the editing environment, allowing developers to request code modifications or explanations without leaving their current file context. This capability also extends to terminal integration, where users can invoke Copilot assistance for command-line tasks and debugging scenarios.

Beyond these core features, the GitHub Copilot extension offers numerous additional capabilities that I did not have the opportunity to explore in detail for this thesis. The extension’s ecosystem is expanding rapidly through frequent feature additions. These new capabilities include autonomous coding workflows, multistep task execution, and integration with external tools via the Model Context Protocol [33]. This accelerated development reflects the broader trend of continuous innovation in AI-powered development tools.

Choice of GitHub Copilot for Software Optimization Testing During the research phase of this thesis, I selected GitHub Copilot as the primary tool to test AI’s capabilities in software optimization tasks. This choice was driven by several considerations:

Since my development work was mainly conducted within VS Code, GitHub Copilot presented an easy integration into my familiar environment. This proved particularly valuable given the rapidly evolving landscape of AI development tools, where new products are released frequently, making it difficult to systematically evaluate all available options.

From a practical perspective, GitHub’s Student Developer Pack [34] currently provides free access to the Copilot Pro plan. This plan includes unlimited code completions and chat interactions with a subset of available models (*Standard Models*), plus 300 monthly requests to the Pro tier’s *Premium Models*.⁸

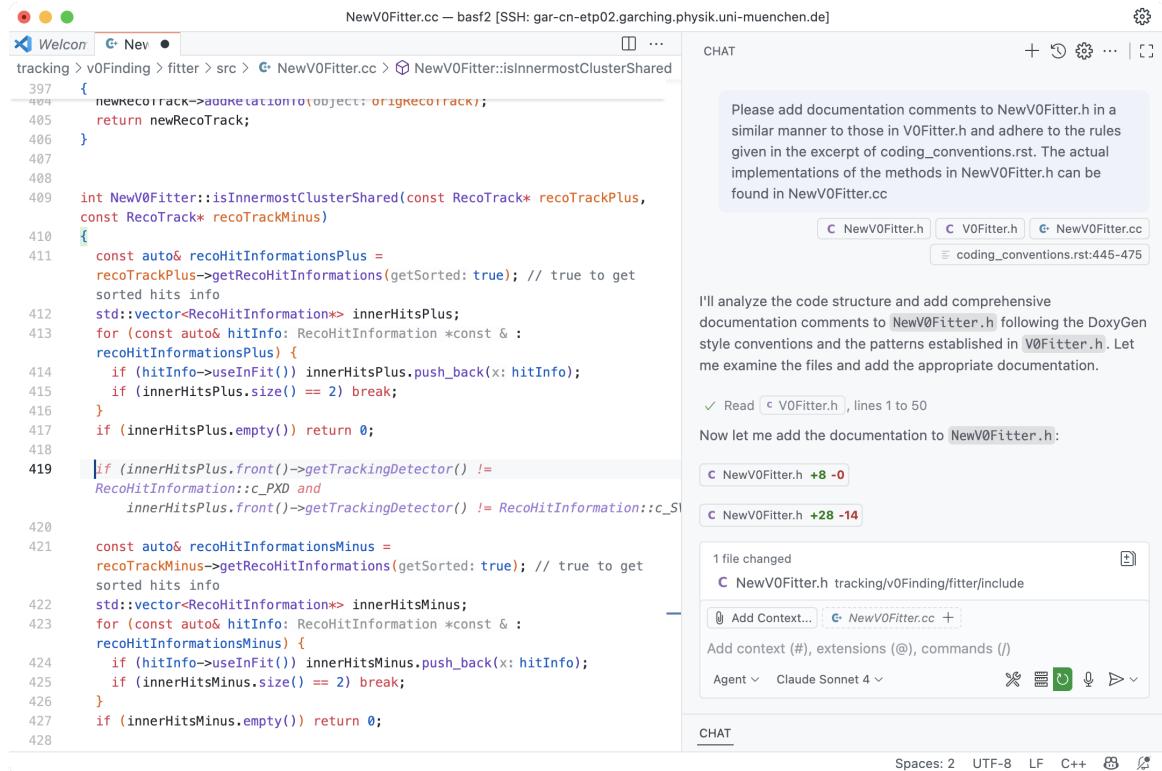


Figure 3: GitHub Copilot integration in VS Code: *Left*: code completion suggestion
Right: Copilot panel with chat mode and model selection; message context is displayed beneath the text

⁸Notes:

- Based on my observations, one request equals one sent chat message
- For a list of all the models available in the Pro tier cf. section 4

2 V0Fitter Optimization

2.1 Introduction

In order to explore to what extent AI can assist developers in complex software ecosystems, the first objective established during the research phase of this thesis was to improve the performance and code quality of basf2's V0Fitter. As it is one of basf2's tracking components with a comparatively high runtime, this presented an ideal target for optimization attempts.

The V0Fitter [35] is a component of the basf2 tracking package concerned with creating V0 particle candidates from reconstructed charged particle tracks.

As detailed in the section of the basf2 documentation on V0 Finding [36] and the Belle II Physics Book [4], the term V0 is used to describe neutral particles that decay into two daughter particles of opposite charge. Due to their relatively long lifetimes, the decay vertices of V0s typically lie outside the beam pipe. As a result, their decay products require special treatment during reconstruction:

In basf2, track parameters are, by default, extrapolated to the interaction point at $(0, 0, 0)$ prior to storage. For particles originating within the beam pipe, the differences introduced by this procedure are typically negligible. However, for particles produced outside the beam pipe, this extrapolation can result in significant deviations between the stored track parameters and the actual values at the point of particle origin. These discrepancies arise primarily due to the curvature of charged particle tracks in the magnetic field, which leads to notable changes in the momentum direction when extrapolated over extended distances. Also, material effects are taken into account during the extrapolation process. As a consequence, particles created outside the beam pipe are subjected to energy loss corrections intended for particles originating within the beam pipe, even though they have not actually interacted with the corresponding material, leading to systematic inaccuracies.

The V0Finder module [37] performs this task for the relevant V0 decays in the context of the Belle II experiment

$$K_S^0 \rightarrow \pi^+ + \pi^- \quad \Lambda^0 \rightarrow p + \pi^- \quad \bar{\Lambda}^0 \rightarrow \pi^+ + \bar{p} \quad \gamma \rightarrow e^+ e^-$$

For a given Belle II event, the V0 reconstruction by the V0Finder module and underlying V0Fitter class proceeds roughly as detailed in Algorithm 1.

Notes on Algorithm 1:

Inner hit detection and removal During the earlier reconstruction steps in basf2, detector hits are grouped, sorted and fitted yielding higher-level track objects representing particle trajectories (see Sec. 1.3, Par. *Reconstruction Pipeline*).

In the context of the V0Finder, the term *inner hits* is used to refer to track hits that lie before the fitted vertex position. Assuming that the particles and their tracks indeed originated from a V0 decay at the vertex, causality implies that the inner hits cannot have been produced by those same particles. For this reason, such detector measurements should not be taken into account during track and vertex fitting as they could skew the fit results.

To detect and remove inner hits in a track, the V0Fitter uses a function provided by the GenFit [38] track fitting toolkit to extrapolate the hit states, i.e. the fitted track parameters at the hit position, to the vertex. The function returns a positive or negative value depending on whether the extrapolation step had a component parallel or antiparallel to the fitted track momentum at the hit position. A positive value therefore implies that the hit lies before the vertex on the trajectory and is taken to be an inner hit.

Given the track hits ordered by their occurrence along the trajectory, for merely detecting inner hits, performing the extrapolation only for the first hit is sufficient. Removing inner hits is done by iterating over the hits, setting the hit to be excluded from subsequent fits if extrapolation returns a value > 0 and breaking out of the loop otherwise (see also Alg. 2).

v0FitterMode [36] The V0Finder configuration parameter v0FitterMode accepts three possible values (0, 1, 2), each controlling the inner hit removal behavior. When set to 0, the V0Fitter stores the V0 candidate immediately after the initial vertex fit without attempting inner hit removal. The default value of 1 enables inner hit removal for detected hits, while a value of 2 applies additional constraints specifically to inner hit removal for SVD detector hits.

Algorithm 1: High-level overview of the V0 reconstruction algorithm

```

function V0FinderModule::event()
    tracksPlus := (track ∈ StoreArray<Track> : track.charge > 0)
    tracksMinus := (track ∈ StoreArray<Track> : track.charge < 0)

    for trkPair ∈ (tracksPlus × tracksMinus)
        ▷ skip precut for nearly parallel tracks since position approximation inaccurate:
        if cos(⟨⟩ trkPair) ≤ precutCosAlpha
            if trkPair.approxVertexPos.rho ≤ precutRho           ▷ filter out decay vertices in beam pipe
                continue

    for v0Hyp ∈ ( $K_S$ ,  $\Lambda$ ,  $\bar{\Lambda}$ ,  $\gamma$ )
        if trkPair.invMassRange ∩ v0Hyp.invMassRange =  $\emptyset$       ▷ cut not performed for  $\gamma$ 
            continue

function V0Fitter::fitAndStore(trkPair, v0Hyp)           ▷ entrypoint of V0Fitter
    vtx := Vertex(0, 0, 0)
    vertexFit(trkPair, v0Hyp, vtx)                          ▷ initial vertex fit

    if fit failed or cut on fitted vertex parameters failed
        return false

    if v0FitterMode = 0                                     ▷ skip inner hit removal
        storeV0(trkPair, v0Hyp, vtx.pos)
        return true

    (trk+, trk-) := trkPair
    while trk+ or trk- has inner hits
        if trk+ has inner hits
            trk+ := removeInnerHitsAndRefit(trk+, vtx.pos, v0Hyp.daughterPlus)

        if trk- has inner hits
            trk- := removeInnerHitsAndRefit(trk-, vtx.pos, v0Hyp.daughterMinus)

        vertexFit((trk+, trk-), v0Hyp, vtx)
        if fit failed or cut on vertex parameters failed
            break

    storeV0((trk+, trk-), v0Hyp, vtx.pos)          ▷ use results of initial fit if removal failed
    return true

```

Vertex fit and cuts The vertex fit is delegated to the RAVE [39] vertex reconstruction package through an interface of GenFit. By default, GenFit configures RAVE to perform the vertex fit using a Kalman filter with smoothing enabled. This process uses the pre-existing fitting information of the positive and negative track to estimate a common point of origin of the tracks, the smoothing algorithm however also refits the track parameters at the vertex.

After a successful fit, the V0Fitter applies cuts to the computed vertex parameters in the following order:

1. ρ of vertex position \geq beamPipeRadius

$\rho := \sqrt{x^2 + y^2}$, radial distance from the solenoid (\approx beam) axis (cf. Fig. 1, right)

beamPipeRadius = 1.0 (inner radius of the beam pipe is 10 mm)

2. χ^2 of vertex fit result \leq vertexChi2CutOutside

3. compute the 4-vectors p_+ , p_- using the RAVE fit results for momentum at the vertex and the known rest mass of the v0Hypothesis particle, require that: $\sqrt{(p_+ + p_-)^2} =: m_{\text{inv.}} \in \text{v0Hyp.invMassRange}$

2.2 Initial Code Analysis and Profiling

It is crucial that any modifications to the V0Fitter code do not alter the physical reconstruction results. It was therefore necessary to first gain a thorough understanding of the processing steps performed by the implementation of the V0Fitter that is currently used by basf2. This also entailed familiarizing myself with various components of the broader basf2 architecture.

To begin, I explored the code's execution path starting from the module entry point V0FinderModule::event, using the IDE's symbol navigation features to trace method calls and variable usage (cf. Sec. 2.6.4). During this process, I documented my findings in pseudocode⁹ to improve my understanding of the algorithmic logic behind the individual routines and data structures.

As a next step, I began profiling the code to identify computationally expensive sections that could be targets for optimization. Initially, I implemented a simple timer class (Lst. 1), whose instances record the time — once during construction and again either upon calling a stop method or alternatively during destruction — and print the time elapsed between both measurements. This enabled the measurement of time spent executing a block of statements (using the explicit stop method) as well as the runtimes of specific functions (automatic destruction on scope exit).

Listing 1: Simple timer class for profiling, a slightly modified version of basf2's Belle2::Utils::Timer [40]

```
#include <sys/time.h>
class MyTimer {
public:
    void stop() {
        get_time(ts1);
        double dt = (ts1.tv_sec - ts0.tv_sec) + (ts1.tv_nsec - ts0.tv_nsec) * 1e-9;
        B2INFO(text << " " << std::scientific << std::setprecision(3) << dt << " s");
        stopped = true;
    }
    explicit MyTimer(const char *text) : text(text) { get_time(ts0); }
    ~MyTimer() { if (not stopped) stop(); }
private:
    void get_time(timespec &ts) { clock_gettime(CLOCK_REALTIME, &ts); }
    timespec ts0, ts1;
    const char *text;
    bool stopped = false;
};
```

I then instrumented the basf2 code by attaching timers at multiple points throughout the V0Finder/Fitter processing flow. Running the instrumented build with a steering script that performs V0 reconstruction¹⁰ yielded a log file that included the timing information.

⁹github.com/philipobi-bachelor/project/artifacts/code-analysis/newv0fitter-pseudocode

¹⁰v0ValidationGenerateSample, see Sec. 2.4 for details

To analyze the log output, I developed a small Python evaluation script¹¹ that iterates over the log lines in reverse, as timers in nested scopes print their results first, grouping the time measurements by event with subgroups for each timer location. For each event, the runtimes of a subgroup are then summed and divided by the measured execution time of V0FinderModule::event. This processing of the extracted log information is done using the Python libraries AwkwardArray [41], NumPy [42] and finally Pandas [43] to average the values and provide further statistical information. The refined data (Tab. 2) provided an initial insight into the distribution of total execution time among the module’s components.

Method		mean	std	min	25%	50%	75%	max
V0FinderModule::event	[ms]	114.4	90.7	0.0	53.0	92.1	153.4	867.5
NewV0Fitter::fitAndStore	[%]	97.4	15.1	0.0	99.7	99.7	99.7	99.9
NewV0Fitter::vertexFit	[%]	90.9	16.8	0.0	88.8	98.4	99.0	99.5
NewV0Fitter::fitGFRaveVertex	[%]	62.9	12.6	0.0	59.8	65.3	69.6	87.1
NewV0Fitter::removeHitsAndRefit	[%]	5.8	9.3	0.0	0.0	0.0	9.3	61.6

Table 2: Initial profiling results; method names are indented to reflect the call hierarchy

However, this custom profiling method lacked the accuracy of more advanced profiling tools and proved overall to be not that well suited for the task at hand.

For instance, it introduced a non-negligible runtime overhead, reducing the reliability of the profiling results. This can be seen from the results of the module timing facility natively built into basf2: the per-event runtime average of the V0Finder with the custom timer instrumentation was significantly higher compared to that of the unmodified version (≈ 114 vs. ≈ 89 ms/call) while the other modules exhibited similar runtimes.¹² This is likely attributable to frequent stream synchronization and flushing by the timers outputting their measured runtimes.

Additionally, due to a misunderstanding, I performed this preliminary profiling on the NewV0Fitter, which is a version of the V0Fitter that already offers significant performance improvements over the original V0Fitter. Despite this, basf2 currently continues to use the initial V0Fitter, since validation indicated a minimal decrease in the NewV0Fitter’s reconstructive performance compared to the V0Fitter.

For the reasons stated above, it was necessary to repeat the profiling with more sophisticated tools both for the NewV0Fitter and especially the original V0Fitter.

Therefore, following the recommendations of Prof. Kuhr and my thesis supervisors, I selected Callgrind as a more reliable and precise alternative. Callgrind [44] is a profiling utility within the Valgrind [45] tools suite. As the Valgrind manual [46] explains, tools within the framework such as Memcheck, Cachegrind and Callgrind instrument a program’s machine code, which is then run on a synthetic CPU provided by the Valgrind core. During code execution, every program instruction is simulated, enabling Callgrind to capture comprehensive data on function call relationships and instruction counts per function call.

A notable consequence of this fine-grained instrumentation however, is a significant execution slowdown of typically one order of magnitude. Callgrind reports results in instruction counts per call rather than wall-clock time, which however serves as an equally valid performance metric as instruction counts are correlated with execution times, especially for similar computational tasks.¹³

Basf2’s --profile <module> option leverages Callgrind functionality to instrument only a specific module¹⁴, while maintaining near-native execution speed for the remaining codebase. Profiling was performed using this option to run a basf2 debug build on a unified script performing V0 reconstruction (cf. Sec. 2.4). After the profiled process exits, Callgrind outputs a text file containing detailed information about function calls and instruction counts (≈ 30 MB and $\approx 3 \times 10^6$ lines of text each).

The gprof2dot [48] utility allows for visualization of the profiling data by converting the Callgrind outputs to call graphs in GraphViz/dot format [49].

¹¹[github/philipobi-bachelor/project/artifacts/profiling/main.ipynb](https://github.com/philipobi-bachelor/project/artifacts/profiling/main.ipynb)

¹²Cf. runtime statistics at the bottom of log outputs for the instrumented build

[github/philipobi-bachelor/project/artifacts/profiling/v0finder-validation.log](https://github.com/philipobi-bachelor/project/artifacts/profiling/v0finder-validation.log) and unmodified build

[github/philipobi-bachelor/validation/validation-newv0fitter-v0FitterMode1/9719fe1/results/current/tracking/v0ValidationGenerateSample.py.log](https://github.com/philipobi-bachelor/validation/validation-newv0fitter-v0FitterMode1/9719fe1/results/current/tracking/v0ValidationGenerateSample.py.log)

¹³Assuming a consistent mix of instruction types, which implies a constant average number of CPU cycles per instruction (CPI), and a fixed CPU clock frequency throughout program execution, the formula CPU time = Instr. count \times CPI \times Clock cycle time [47] indicates that execution time is directly proportional to the instruction count.

¹⁴Cf. macro CALL_MODULE in basf2/framework/core/src/EventProcessor.cc

The resulting graphs provide valuable information:

- The percentages in each function node represent the proportion of the total instruction count executed within that function including (excluding) nested calls (referred to as inclusive and exclusive cost in the following)
- The nodes' color-coding indicates their inclusive cost
- The edges connecting the function nodes show the distribution of inclusive self cost among the callees of a given function, as well as the total number of times a child function was called by its parent

Note: Even with the default pruning settings of gprof2dot (node and edge inclusive cost thresholds of 0.5% and 0.1%, respectively), the generated call graphs remained very large — typically on the order of 300 nodes. To keep the figures in this thesis readable and focused, only smaller segments of the call graph are shown, highlighting relevant parts of the call structure.

These segments were created by using custom pruning thresholds and manually limiting the graph depth. Graph depth can be limited with gprof2dot options that restrict the call graph either from a specific parent node downward or from a child node upward. To further improve clarity, unnecessary details were removed: template and type information were stripped using the gprof2dot --strip option, and a custom script¹⁵ was used to remove less important data such as the shared library name and the total number of calls per function.

The complete call graphs generated with gprof2dot's default pruning settings, along with the raw profiling data obtained from Callgrind, are provided in a repository for reference.¹⁶

I profiled the V0Finder twice using a unified profiling script¹⁷, once with the V0Fitter and once with the NewV0Fitter as fitting backend. The call graphs in Figure 4 visualize the collected data and Table 3 presents some additional metrics extracted from the profiling output files using Valgrind's callgrind_annotate command.

Method	V0Fitter			NewV0Fitter		
	Calls	Instr.	I%/Call	Calls	Instr.	I%/Call
TrackFitter::fit	252	3.42×10^{10}	0.1561	28	3.54×10^9	0.3312
fitGFRaveVertex	1391	1.69×10^{10}	0.0140	1183	1.44×10^{10}	0.0318
copyRecoTrack	286	1.90×10^9	0.0077	28	1.95×10^8	0.0182
extrapolateToPoint	3216	1.67×10^{10}	0.0060	806	4.78×10^9	0.0156
genfit::Track::Track	2782	1.43×10^{10}	0.0059	2366	1.27×10^{10}	0.0141
Total Instr. Count		8.69×10^{10}			3.81×10^{10}	

I%/Call: Inclusive cost percentage per function call, i.e. $\frac{\text{Instr.}}{(\text{Total Instr. Count}) \times \text{Calls}} \times 100$

Table 3: Profiling data for selected function calls, ordered by cost per call

The refined data provides several valuable insights into the module's performance characteristics:

- Runtime overhead is almost entirely concentrated in the fitter modules, the cost incurred by computations in the V0Finder itself is negligible (Fig. 5)
- Track and vertex fitting procedures have the highest computational cost per call (Tab. 3)
- The performance improvement of the NewV0Fitter over the V0Fitter can be primarily attributed to a significant reduction in track fitting operations, extrapolation of track parameters to the vertex and RecoTrack copying (Fig. 4, Tab. 3). For the underlying reasons for these savings cf. Section 2.3
- The parts of the call graph corresponding to the RAVE vertex fitting process (Fig. 6) show that computational load is broadly distributed across many different functions. The initial profiling data for the NewV0Fitter (Tab. 2) had indicated that this was a costly operation, making it a candidate for optimization, however navigating the complex call hierarchy of the RAVE code proved to be challenging. As the more thorough results of Callgrind profiling indicate, no single function within the vertex fitting process emerges as a clear bottleneck or an obvious target for optimization

¹⁵[github/philipobi-bachelor/profiling/dotfmt.py](https://github.com/philipobi-bachelor/profiling/dotfmt.py)

¹⁶Cf. graph.svg and callgrind.out files in [github/philipobi-bachelor/profiling](https://github.com/philipobi-bachelor/profiling)

¹⁷For implementation details refer to section 2.4

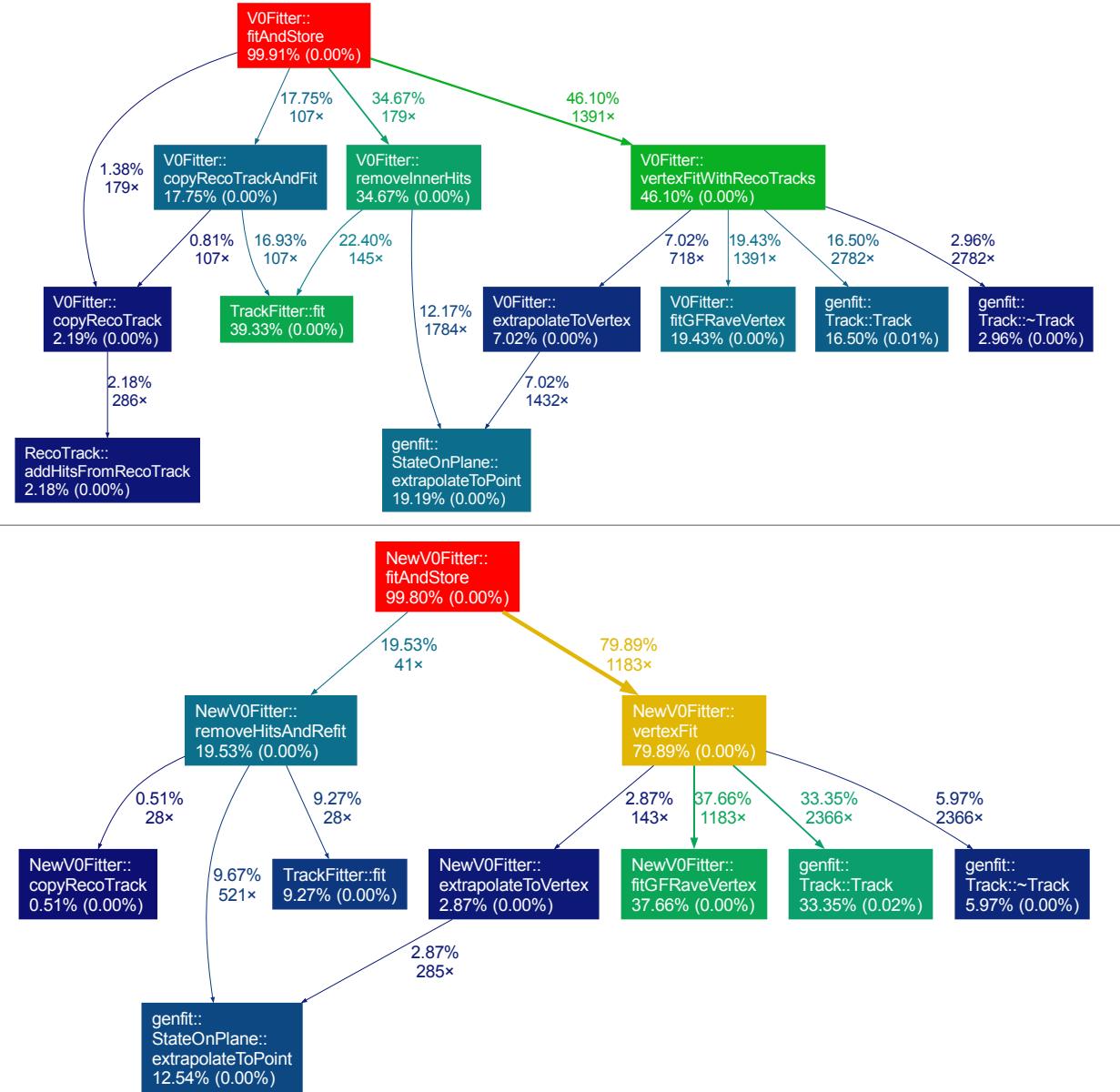


Figure 4: Call graph segments starting from the fitAndStore class entry point, annotated with profiling data. *Top:* V0Fitter, *Bottom:* NewV0Fitter. Generated using gprof2dot and GraphViz.

- The profiling data also highlights a significant overhead from genfit::Track construction (Fig. 4, Tab. 3), particularly its copy constructor as evident in the unstripped call graph¹⁸

2.3 In-depth Code Analysis and Refactoring

Based on the insights gained from profiling and considering that the original V0Fitter remains the fitter implementation currently used by the V0Finder, I was able to more clearly define an objective: refactoring the V0Fitter code while exploring potential applications of AI throughout the process. Specifically, the goal was to modularize the code, reducing monolithic structures and eliminating redundancy as well as improving the performance while preserving the overall logic and reconstruction accuracy of the original implementation.

I started by analyzing the V0Fitter source code and documented my findings using pseudocode as I had previously done for the NewV0Fitter¹⁹. Then, with a solid understanding of both fitter implementations, I was ready to begin the actual refactoring. To aid in the redesign, I prompted several LLMs for conceptual strategies on how to refactor the code, particularly regarding the decomposition of the V0Fitter’s tasks and data structures into more cohesive, self-contained components (cf. section 2.5.1). Once I had established a preliminary

¹⁸See for example github.com/philipobi-bachelor/profiling/profiling-v0fitter/050e500/graph.svg

¹⁹github.com/philipobi-bachelor/project/artifacts/code-analysis/v0fitter-pseudocode

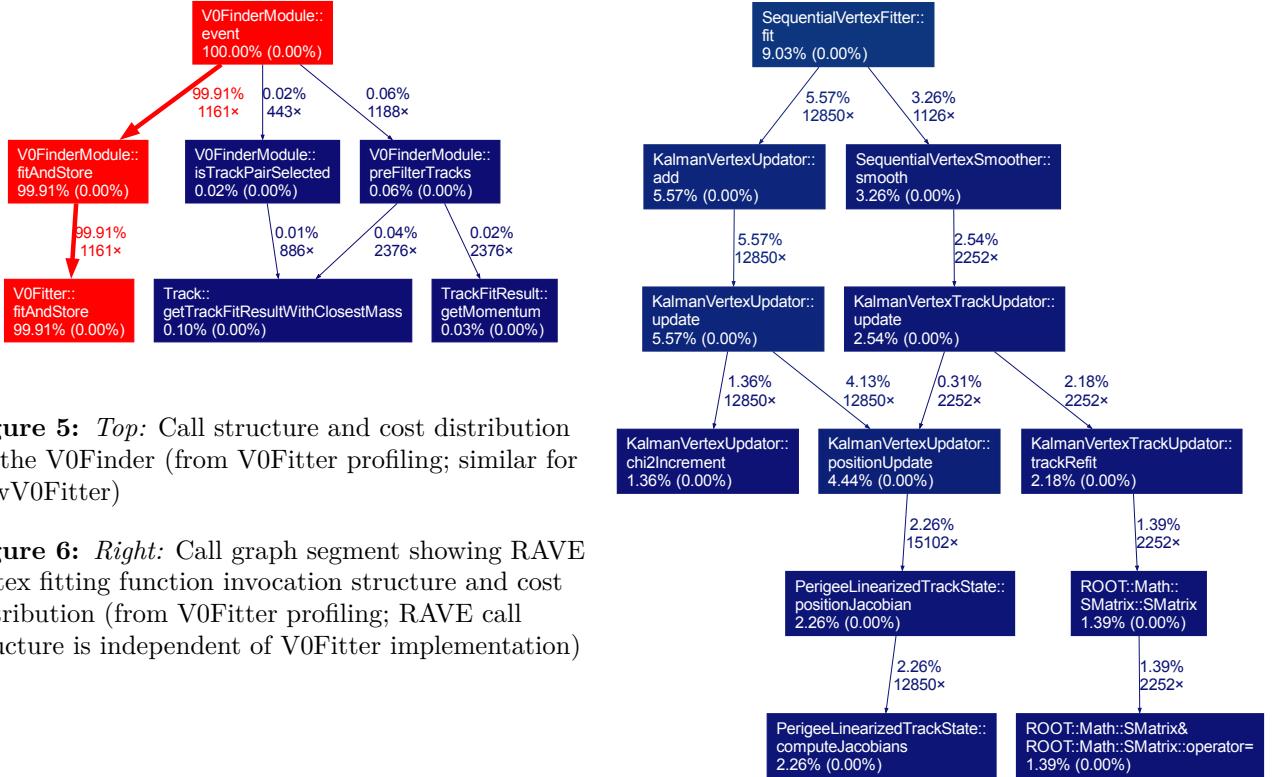


Figure 5: Top: Call structure and cost distribution for the V0Finder (from V0Fitter profiling; similar for NewV0Fitter)

Figure 6: Right: Call graph segment showing RAVE vertex fitting function invocation structure and cost distribution (from V0Fitter profiling; RAVE call structure is independent of V0Fitter implementation)

architectural outline for the refactored version, incorporating some of the AI’s suggestions, I proceeded with the implementation.

2.3.1 Short background Information and Class Explanations

The following gives a short overview of important classes in the context of V0 fitting, summarizing information from the respective class documentations [50][51][52][53]:

Belle2::RecoTrack

- Provides an interface to access fitted track parameters and hit information associated with the track
- Fitting of RecoTracks is performed using the TrackFitter class
- Instances exist only during the reconstruction stage, afterwards, only the more lightweight Belle2::Track objects remain containing references to fit results of the track for multiple particle hypotheses
- Useful wrapper around a genfit::Track

genfit::Track

- Described in documentation as “the most holy part of the RecoTrack”
- Represents a track within the GenFit fitting library, contains all relevant track fitting information such as fitted track parameters and their covariances
- Usually holds multiple track representations, corresponding to different particle hypotheses, which can be fitted in parallel
- Essentially, when fitting a RecoTrack, the TrackFitter adds the hits selected for the fit (as represented by their `useInFit` property) to the genfit::Track²⁰ and then invokes a GenFit method to perform the actual track fitting

²⁰Cf. `TrackFitter::fit` and its invocation of `MeasurementAdder::addMeasurements` [54]

Belle2::V0

- Stored to the DataStore for use in the analysis stage, encapsulates information about a reconstructed V0 candidate:
 - Pointers to the pair of Belle2::Tracks used for reconstructing the V0
 - Pointers to the fit results of the specific track representations that correspond to the V0 daughter particle hypotheses
 - Fitted vertex position

2.3.2 Overview of Performed Optimizations

Throughout the refactoring process, I compared the steps both existing fitter versions took to perform a specific task, trying to identify opportunities to remove unnecessary operations or implement more efficient solutions without altering the underlying logic.

This allowed me to identify several key areas for optimization in the code and implement the logic more efficiently in the refactored version:

RecoTrack Copying For each hit removal iteration, both the V0Fitter and the NewV0Fitter use a similar procedure to attempt the removal of detected inner hits in the positive and negative RecoTrack. A simplified overview of this process is presented in Algorithm 2 for the V0Fitter, with the NewV0Fitter employing a largely similar method.

Algorithm 2: Pseudocode for hit removal procedure in the V0Fitter

```

zip :  $((a_1, \dots, a_N), (b_1, \dots, b_N)) \mapsto ((a_1, b_1), \dots, (a_N, b_N)), N \in \mathbb{N}$ 

function V0Fitter::removeInnerHitsAndRefit(track, vertexPos, particleHyp)
   $\triangleright$  NewV0Fitter: only copy and modify a bool vector of useInFit flags at first
  trackCopy := copyRecoTrackWithHits(track)
  for (hitCopy, hit)  $\in$  zip(trackCopy.hits, track.hits)
    if not hit.useInFit
      hitCopy.useInFit := false
      continue

    if extrapolateToPoint(hit.state, vertexPos)  $>$  0
      hitCopy.useInFit := false
    else
      break

   $\triangleright$  NewV0Fitter: copy track here if more hits removed, apply useInFit vector to the copy
  TrackFitter::fit(trackCopy, particleHyp)
  return trackCopy

```

In addition to copying all associated hit information, copying a RecoTrack also involves creating a new genfit::Track instance within the copied RecoTrack. To optimize this procedure while keeping the original RecoTrack unmodified, the refactored version creates a copy exactly once, before the first hit removal attempt and reuses this copy for all subsequent iterations.

genfit::Track Copying To perform the vertex fit, pointers to the genfit::Track objects of the RecoTrack pair must be passed to the vertex fitting interface. To prevent modification, both existing implementations copy-construct the genfit::Tracks before the fit. This frequent copy-construction explains the significant number of instructions spent in the genfit::Track constructor that were observed during profiling (Tab. 3).

After reviewing the GenFit-RAVE interface code, I concluded with high confidence that the vertex fitting procedure does not modify the genfit::Tracks:

First of all, the genfit::Tracks are converted to rave::Tracks before being passed to the RAVE vertex fitter, a process during which all necessary data is copied.²¹ Secondly, the only track state that is modified during this process is a copy of the original object, as explained in the inline comment of the relevant source code line.²²

Therefore, the only modification made to the genfit::Track during vertex fitting is the setting of its cardinal representation, which allows to select one of the multiple track representations in the genfit::Track as main version. Only the cardinal representation is passed to RAVE for vertex fitting, therefore it needs to be set to that of the daughter particle hypothesis to obtain accurate physical results.

To address this concern in the refactored version while avoiding copying, I implemented functionality to restore the cardinal representation of the original RecoTrack's genfit::Track to its initial state as soon as the object is no longer needed by the fitter.²³ This optimization eliminates the need to create copies of genfit::Tracks altogether, especially for the initial vertex fit. The initial vertex fit is the most frequently performed fitting procedure, since for most calls to fitAndStore, it either fails or no inner hits are detected.

Although each individual copy operation is not exceedingly expensive, eliminating them entirely saves a sizeable amount of instructions (Tab. 4).

Track Fitting for Unpromising Hypotheses As track refitting is the most costly operation per call in terms of instructions, reducing unnecessary calls to TrackFitter::fit presents the most significant opportunity for optimization.

This is also the primary area where the NewV0Fitter achieves its performance advantage over the V0Fitter as evident from the profiling data (Tab. 4). It accomplishes this by enforcing the invariant mass cut after a vertex fit, regardless of whether an inner hit was detected (see Par. *Vertex fit and cuts* in Sec. 2.1 for details on the performed cut).

The V0Fitter, in contrast, only applies this cut if no inner hits are found, or if it is configured to ignore inner hits. This allows many candidates with invariant masses far outside the allowed range for the hypothesis to enter the iterative hit removal and refitting process, only for the invariant mass cut to fail once inner hits have been removed, resulting in numerous wasteful track refitting calls.

In the refactored version, I adopted the strategy of consistently enforcing the invariant mass cut after the initial vertex fit, while adding a tolerance to the allowed range if inner hits were detected.

V0 Object Stashing In the V0Fitter, if an initial vertex fit is successful but inner hits are present and hit removal is enabled, the vertex fit results are discarded and the hit removal process begins. If the subsequent hit removal fails, the initial vertex fit is repeated with the original RecoTracks and the V0 object is stored regardless of inner hits.

To avoid the redundant vertex fit in such cases, I established functionality to construct and temporarily stash the Belle2::V0 object after the initial vertex fit, allowing it to be retrieved and immediately stored should the hit removal process not be successful. The NewV0Fitter implements a similar feature, but with the key difference that it stores the result of the last successful vertex fit, which may not be the initial one.

2.3.3 Overview of Refactored Code Architecture

Note: The refactored code overwrites the previous NewV0Fitter implementation in NewV0Fitter.h/.cc²⁴ to simplify integration and avoid changes to existing infrastructure like build scripts and configuration options (such as the V0Finder boolean configuration option useNewV0Fitter). This can be easily changed by renaming the relevant files and the NewV0Fitter namespace, where the refactored implementation is defined, should my implementation be merged into the official basf2 branch.

The refactored V0Fitter code replaces the previously monolithic Fitter class with several distinct components or-

²¹Cf. GenFit source code in basf2 for [GFRaveVertexFactory::findVertices](#) and its nested invocations of [genfit::GFTackToTrack](#)

²²See the GenFit source code in basf2: [basf2/genfit2/code2/GFRave/src/GFRaveConverters.cc:68](#)

²³Handled by the destructor of a wrapper class; this is briefly touched upon in Lst. 7, (2), for details see the source code implementation

²⁴Located at basf2/tracking/v0Finding/fitter/{include/NewV0Fitter.h, src/NewV0Fitter.cc}

ganized within the NewV0Fitter namespace. The following section provides a brief overview of the most relevant classes and members. For further implementation details, see the source code of the refactored V0Fitter.²⁵

NewV0Fitter::Fitter This is the main class orchestrating the fitting process, providing the fitAndStore method that serves as the primary interface to the V0Finder, consistent with the two previous fitter implementations (Lst. 2).

NewV0Fitter::V0 Instances of this class are created upon entry into fitAndStore, requiring the V0 hypothesis of the current fitting operation as argument (Lst. 2). The class encapsulates hypothesis-specific logic, such as applying the invariant mass cut, and managing the construction, stashing and storage of Belle2::V0 objects (Lst. 3). It also provides intermediate storage for data relevant to V0 reconstruction, such as the daughter particle types for a given hypothesis, the vertex fit results, and stashed Belle2::V0 objects.

NewV0Fitter::Track Two NewV0Fitter::Track objects are constructed at the start of the fitAndStore function, corresponding to both the positive and negative Belle2::Track passed by the V0Finder (Lst. 2). The class provides methods to check for inner hits and attempt their removal, while managing the copying of underlying RecoTracks to ensure the original objects are never modified (Lst. 4).

NewV0Fitter::Config This struct holds all fitter configuration options as const members, such as the invariant mass ranges for the specific V0 hypotheses, fitter mode, and DataStore array names (Lst. 5). An instance is created and configured by the V0Finder during initialization and passed to the fitter's constructor which assumes ownership of the object. Static constant pointers to the Config instance are provided to certain other components of the namespace.

This for example allows the NewV0Fitter::V0 to look up the invariant mass range for a certain v0Hypothesis, and enables the NewV0Fitter::Track to get the name of the StoreArray containing the RecoTracks, and subsequently retrieve the RecoTrack corresponding to the Belle2::Track passed by the V0Finder.

2.4 Evaluating Computational and Reconstruction Performance of the V0Fitter Implementations — Methodology and Results

2.4.1 Data Generation, Validation and Profiling Steering Scripts

To validate the V0Fitter implementations, I used a slightly modified version of the basf2 steering script v0ValidationGenerateSample.py [55] (cf. Paragraph *Modifications of the Validation Script*).

Input Data v0ValidationGenerateSample.py requires a ROOT file as input, which is generated by the steering script KShortGenSimNoBkg.py [56]. This script performs a Monte Carlo simulation of 1500 events without beam background overlay, providing clean signal data for validation purposes.

In each event, five KShort particles with uniformly distributed momentum magnitude and angular parameters are generated at the interaction point using the basf2 ParticleGun module [57]. Then, the particle interactions with the detector components, as well as the resulting detector response measurements are simulated. This simulated measurement data is stored to KShortGenSimNoBkg.root, which serves as input for v0Validation GenerateSample.py.

Validation Script The script v0ValidationGenerateSample.py first performs track reconstruction on the generated data in KShortGenSimNoBkg.root. As a next step, it reconstructs V0 candidates using the V0Finder module and matches them to the true KShorts based on the simulation's particle identity information. Finally, for each simulated KShort that decayed to pions, the script stores the simulated parameters (commonly referred to as the MC truth variables) and, if available, the parameters of a matched reconstructed V0 candidate to the output file V0ValidationHarvested.root.

²⁵The refactored implementation is available in the corresponding files on the v0fitter-refactor branch of the forked basf2 repository [github/philipobi-bachelor/basf2:v0fitter-refactor](https://github.com/philipobi-bachelor/basf2:v0fitter-refactor):
[github/philipobi-bachelor/basf2:v0fitter-refactor/tracking/v0Finding/fitter/include/NewV0Fitter.h](https://github.com/philipobi-bachelor/basf2:v0fitter-refactor/tracking/v0Finding/fitter/include/NewV0Fitter.h) and
[github/philipobi-bachelor/basf2:v0fitter-refactor/tracking/v0Finding/fitter/src/NewV0Fitter.cc](https://github.com/philipobi-bachelor/basf2:v0fitter-refactor/tracking/v0Finding/fitter/src/NewV0Fitter.cc)

Listing 2: Relevant members of the Fitter class in the refactored V0Fitter implementation

```

namespace Belle2::NewV0Fitter {
class Fitter {
public:
    Fitter(const Config& config);
    bool fitAndStore(const Belle2::Track* trackPlus,
                     const Belle2::Track* trackMinus,
                     const Const::ParticleType& v0Hypothesis, bool& isForceStored,
                     bool& isHitRemoved);

private:
    bool fitVertexAndPrecut(V0& v0, Track& trackPlus, Track& trackMinus);
    const Config m_config;
};

}

namespace Belle2::NewV0Fitter {
bool Fitter::fitAndStore(const Belle2::Track* trackPlus_,
                         const Belle2::Track* trackMinus_,
                         const Const::ParticleType& v0Hypothesis,
                         bool& isForceStored, bool& isHitRemoved) {
    V0 v0 = V0::create(v0Hypothesis);
    const auto& [particleHypPlus, particleHypMinus] = v0.getDaughterParticles();
    Track trackPlus(true, trackPlus_, particleHypPlus);
    Track trackMinus(false, trackMinus_, particleHypMinus);

    // precut and perform initial vertex fit
    bool status = fitVertexAndPrecut(v0, trackPlus, trackMinus);
    if (!status) return false;
    bool passInvMassCut = v0.invMassCut();

    // check for inner hits, hit removal, etc.
    // ...
}
}

```

Listing 3: Relevant members of the V0 class in the refactored V0Fitter implementation

```

namespace Belle2::NewV0Fitter {
class V0 {
public:
    bool invMassCut(const uint tolerance = 0) const;
    void stash(V0ObjPair&& v0ObjPair);
    static V0 create(const Const::ParticleType& v0Hypothesis);

private:
    std::optional<V0ObjStash> m_v0ObjStash;
    const Const::ParticleType& m_V0Particle;
    const std::tuple<double, double>& m_V0InvMassRange;
    static const Config* s_config;
};
}

```

Listing 4: Relevant members of the Track class in the refactored V0Fitter implementation

```
namespace Belle2::NewV0Fitter {
class Track {
public:
    Track(const bool isTrackPlus, const Belle2::Track* track,
          const Const::ChargedStable& particleHyp);
    bool checkInnerHits(const ROOT::Math::XYZVector& vertexPos);
    bool removeInnerHitsAndRefit(const ROOT::Math::XYZVector& vertexPos);
private:
    const Belle2::Track* m_track;
    static const Config* s_config;
};
}
```

Listing 5: Relevant members of the Config class in the refactored V0Fitter implementation

```
namespace Belle2::NewV0Fitter {
struct Config {
public:
    const std::string arrayNameRecoTracks;
    const std::string arrayNameCopiedRecoTracks;
    const std::tuple<double, double> invMassRangeKshort;
    const std::tuple<double, double> invMassRangeLambda;
    const std::tuple<double, double> invMassRangePhoton;
    const double beamPipeRadius;
    const double vertexChi2CutOutside;
    const int v0FitterMode;
};
}
```

v0ValidationCreatePlots.py [58] generates a variety of validation plots from the data in V0ValidationHarvested.root and stores them to V0Validation.root. These include histograms of reconstructed physical quantities, e.g. the invariant mass of the KShort, and their residua²⁶, as well as plots of the reconstruction efficiency²⁷ as a function of variables such as V0 momentum and decay position (cf. Fig. 9).

Comparison plots The script b2validation-plots [59] allows to compare validation results from different code revisions. Using the script’s -r option, multiple revision directories — each containing validation outputs such as V0Validation.root — can be specified for comparison, with the first provided revision being treated as the reference. b2validation-plots then overlays results from all selected revisions in comparison plots and calculates consistency metrics (e.g. χ^2), color-coding the plots accordingly.

The script expects all revisions to be located in the same parent directory. Since my directory structure did not meet this requirement, I implemented a helper script²⁸ that symlinks selected validation results into a temporary directory with descriptive names (as they are used in the plot labels), runs b2validation-plots, collects the relevant outputs (PDF plots and consistency data) and then deletes the temporary directory.

Reproducibility and Determinism in Processing Results Although KShortGenSimNoBkg.py sets a random seed at the start of its execution, slight fluctuations in the generated data have been observed. To improve reproducibility, I generated the KShortGenSimNoBkg.root file once and used it as a consistent input for all subsequent validation runs.

Similarly, v0ValidationGenerateSample.py also initializes a random seed before processing. However, even with the unified use of the KShortGenSimNoBkg.root file, the validation results did not appear to be fully deterministic. For instance, the number of V0s reconstructed by the V0Finder fluctuated slightly when comparing the results of consecutive v0ValidationGenerateSample.py executions.²⁹

The source of this randomness is currently under investigation and is expected to be addressed in future versions of basf2. It is important to note that the observations described here were made using basf2 at the state of the release-09-00-00 commit and may not apply to earlier or later versions of the software.

Modifications of the Validation Script To enable validation of the different fitter implementations, I made slight modifications to the v0ValidationGenerateSample.py script. First, as mentioned earlier, the input file path was updated to an absolute path pointing to the unified KShortGenSimNoBkg.root file. Additionally, to validate non-default V0Finder configurations — such as using the NewV0Fitter as the fitting backend instead of the default V0Fitter, or testing v0FitterModes 0 and 2 — the relevant parameters within v0ValidationGenerate Sample.py were adjusted. Aside from these changes, the script remained unaltered.

Custom Profiling Script To profile the three fitter versions, I used the script v0ValidationGenerate Sample.py as a foundation and made significant modifications. To begin with, the number of processed events was reduced from 1500 to 20. This adjustment was necessary since Callgrind profiling instrumentation drastically decreased execution speed³⁰, which made processing all events prohibitively time-consuming. Additionally, I removed the matching of Monte Carlo V0s with their reconstructed counterparts and subsequent storage of their data to V0ValidationHarvested.root, as this process is unrelated to the actual fitter processing and thus only adds unnecessary runtime overhead during profiling. To enable testing of each implementation, I set the use-NewV0Fitter parameter in the script according to the specific fitter being profiled. However, the v0FitterMode parameter was left at its default value of 1.

2.4.2 Testing Methodology

To make profiling and validation results reproducible across different code versions, I adopted the following approach:

²⁶Given a physical quantity Q and a Monte Carlo V0 with a matched reconstructed V0, the residual is defined as

$$Q_{\text{res.}} := Q_{\text{rec.}} - Q_{\text{MC}}$$

²⁷Efficiency := nReconstructed / nTotal

²⁸[github/philipobi-bachelor/validation/validation-plots.py](https://github.com/philipobi-bachelor/validation/validation-plots.py)

²⁹Number of stored V0s from three consecutive v0ValidationGenerateSample.py executions, using the same machine and basf2 executable: 5896, 5895, 5894; cf. validation logs in [github/philipobi-bachelor/validation/validation-v0fitter-v0FitterMode1/e23d80d/runs](https://github.com/philipobi-bachelor/validation/validation-v0fitter-v0FitterMode1/e23d80d/runs)

³⁰E.g. processing time for one event ≈ 100 s instead of ≈ 660 ms in optimized build

- Each profiling or validation run was performed on its own branch in the basf2 repository, derived from the common release-09-00-00 commit (see Fig. 7)
 - Profiling and validation branches contain the unified steering scripts described in the previous paragraph, the only differences between individual branches are that relevant parameters were adjusted per branch: for example the unified validation script on branch validation-newv0fitter-v0fittermode2 configures the V0FinderModule with useNewV0Fitter: True and v0FitterMode: 2
 - In order to test the effect of code modifications, the relevant development commit is first merged into the respective testing branches (cf. Fig. 7), then the branch containing the script to be run is checked out, the binaries recompiled if necessary, and finally basf2 is run using the branch’s steering script.
- Note:* this procedure was only necessary for testing the V0Fitter refactor, as the existing implementations of V0Fitter and NewV0Fitter were not modified
- Profiling and validation results were saved in directories named after the shortened hashes of the commits they were generated with and are documented in dedicated repositories³¹

This strategy ensures that test results are clearly linked to specific code versions, allowing for straightforward comparison across versions.

In the following section, the short hashes in tables and images displaying validation and profiling results each refer to the commit which was used to generate the data.

2.4.3 Comprehensive Testing Results for the V0Fitter Implementations

Throughout the refactoring process, I continuously compared the refactored V0Fitter’s performance with that of the two original implementations, which enabled me to assess how individual modifications affected computational efficiency and reconstruction quality. This iterative evaluation shaped my coding approach, prompting me to revise or discard some optimization efforts before arriving at the current implementation described in the previous section.

Comparing profiling results of the refactored version and the prior implementations reveals a significant decrease in total executed instructions, primarily due to elimination of genfit::Track copy constructor invocations and fewer calls to TrackFitter::fit compared to the original V0Fitter implementation (Fig. 8, Tab. 4).

The performed optimizations also result in noticeable speedups when running the full validation scripts on optimized builds (Tab. 5).

Method	V0Fitter (050e500)			NewV0Fitter (3ca6556)			V0Fitter Refactor (a4aaaf96)		
	Calls	Instr.	I%/Call	Calls	Instr.	I%/Call	Calls	Instr.	I%/Call
TrackFitter::fit	252	3.42×10^{10}	0.1561	28	3.54×10^9	0.3312	49	6.43×10^9	0.5018
fitGFRaveVertex	1391	1.69×10^{10}	0.0140	1183	1.44×10^{10}	0.0318	1189	1.10×10^{10}	0.0353
copyRecoTrack	286	1.90×10^9	0.0077	28	1.95×10^8	0.0182	50	3.19×10^8	0.0244
extrapolateToPoint	3216	1.67×10^{10}	0.0060	806	4.78×10^9	0.0156	1745	8.06×10^9	0.0177
genfit::Track::Track	2782	1.43×10^{10}	0.0059	2366	1.27×10^{10}	0.0141	0	0.00	0.0000
Total Instr. Count	8.69×10^{10}			3.81×10^{10}			2.62×10^{10}		

Table 4: Comprehensive Profiling Metrics for the V0Fitter Implementations

³¹[github/philipobi-bachelor/profiling](https://github.com/philipobi-bachelor/profiling) and [github/philipobi-bachelor/validation](https://github.com/philipobi-bachelor/validation)

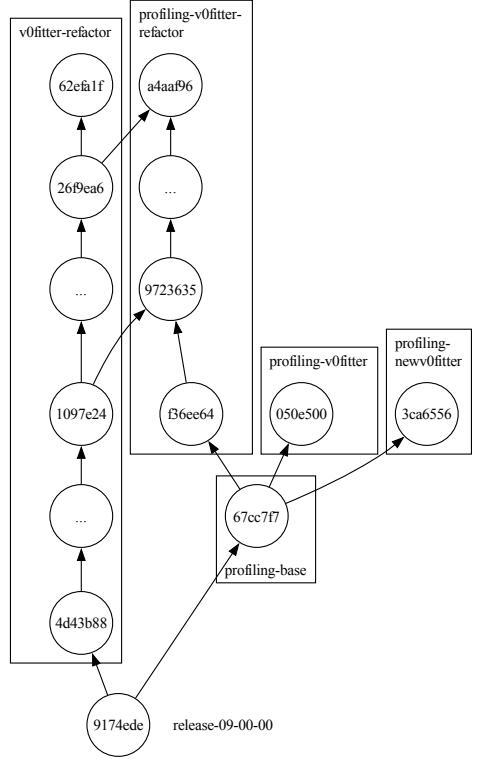


Figure 7: Testing branch structure, illustrated for the profiling branches. The validation branches have similar structure, however there is an additional dimension of subdivision based on the v0FitterMode value

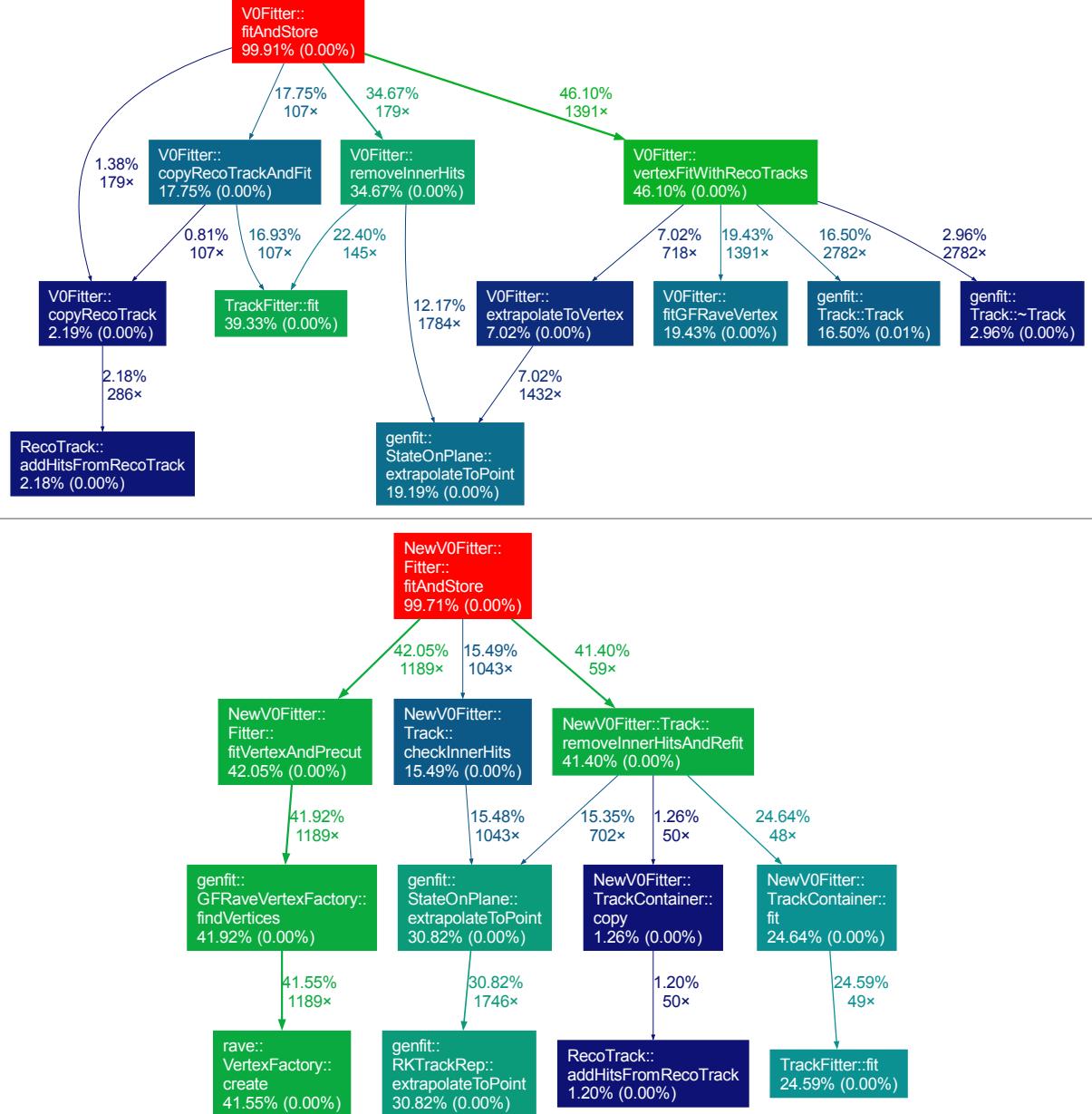


Figure 8: Call graph segments starting from the `fitAndStore` class entry point, annotated with profiling data. *Top:* V0Fitter, *Bottom:* Refactored V0Fitter

Judging by the validation results, i.e. comparing the statistics on the stored V0s and generating plots from the collected ROOT data, the refactored V0Fitter maintains a reconstruction efficiency similar to that of the previous versions. The number of stored V0s for a given fitter configuration is relatively similar across the different implementations (Tab. 5), and the plots, using the original V0Fitter data as reference, match closely (Fig. 9).

The tolerance parameter introduced for the invariant mass cuts in the refactored version allows to control the number of track pairs considered for inner hit removal after the initial vertex fit. By adjusting this value, it is possible to balance between the highest computational efficiency (tolerance = 0) and the greatest reconstruction sensitivity ($\text{tolerance} \gg 1$). Since the inner hit removal has the highest computational overhead in the V0 finding process per call (4), it appears worthwhile to investigate, whether more advanced heuristics can be developed to better judge how promising a V0 candidate is after the initial vertex fit.

2.5 Documentation of AI Usage — Methods and Experiences

From the models available in GitHub Copilot, I selected the following for comparison³²:

³²See Section 4 for model details and selection rationale

Impl. / v0FitterMode		Stored V0s			Time [ms] / call		Commit
		Total	Detected	Removed			
V0Fitter		5849	0	0	82.31 ± 62.50		66d35ef
NewV0Fitter	0	5848	0	0	78.16 ± 58.96		05c5b32
V0Fitter Refactor		5873	0	0	42.38 ± 34.58		6e2dcd6
V0Fitter		5895	1364	1002	165.11 ± 157.20		e23d80d
NewV0Fitter	1	5846	1662	1317	89.10 ± 71.48		9719fe1
V0Fitter Refactor		5878	1352	1010	70.77 ± 74.86		1492106
V0Fitter		5896	1365	997	163.11 ± 154.88		54bcd16
NewV0Fitter	2	5849	1669	1317	88.86 ± 71.48		8b68401
V0Fitter Refactor		5879	1355	979	71.42 ± 75.90		cd1f30d

Table 5: Comprehensive Validation Metrics for the V0Fitter Implementations

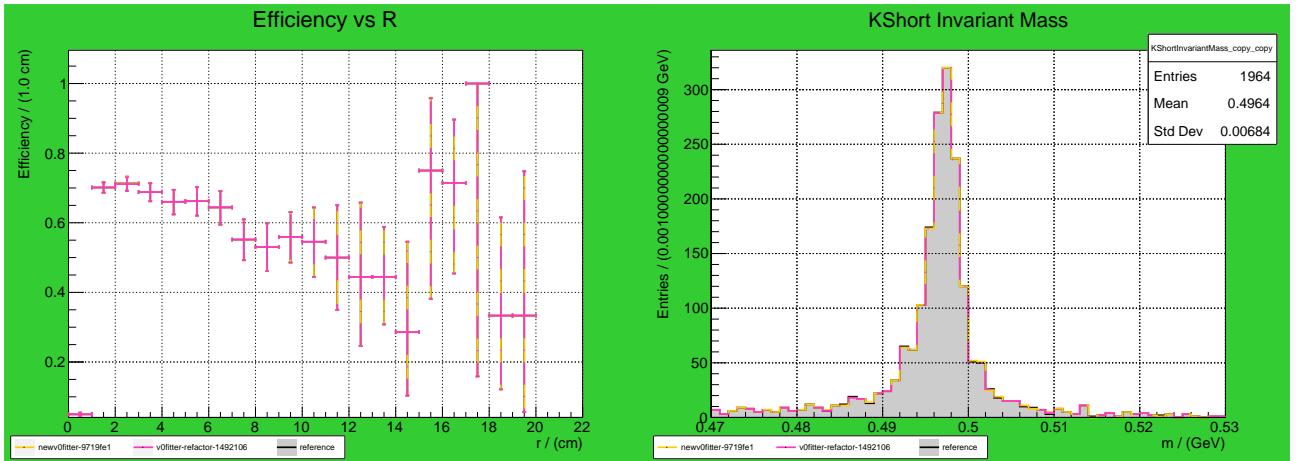


Figure 9: Plots comparing the validation outputs of the V0Fitter (labelled *reference*, commit 050e500), NewV0Fitter and refactored V0Fitter

- Anthropic’s Claude Sonnet 4 [60]
- Gemini 2.5 Pro by Google [61]
- OpenAI o4-mini [62]
- OpenAI’s GPT-4o [63]

Note: Each of the following three sections summarizes key findings from testing the models’ capabilities on a task related to the V0Fitter refactoring process. For further details, such as metaprompts and complete chat logs — including the exact prompts, model responses and any generated code edits³³ — please refer to the corresponding subdirectory of the chat archive repository³⁴. A link to the relevant subdirectory is provided in the footnote of each section heading.

2.5.1 Refactoring advice³⁵

Before beginning with the refactoring of the V0Fitter, I prompted the models for general refactoring advice. I provided all models with the same prompt and included the V0Fitter source code³⁶ as context, using the Ask mode in VS Code’s Copilot Chat panel.

³³Cf. “Squashed changes” section at the end of the chat logs

³⁴[github/philipobi-bachelor/chats](https://github.com/philipobi-bachelor/chats)

³⁵[github/philipobi-bachelor/chats/v0fitter/refactoring-advice](https://github.com/philipobi-bachelor/chats/v0fitter/refactoring-advice)

³⁶Basf2 repository at commit release-09-00-00

Overall, the responses demonstrated a reasonable comprehension of the code architecture. However, some models tended to excessively decompose the V0Fitter into individual components: for instance o4-mini suggested a dedicated TrackFitResultFactory solely for creation of TrackFitResult objects, as well as a TrackHypothesisResolver class with the singular responsibility of mapping V0 to daughter hypotheses.

Ultimately, I chose to adopt several of the suggestions during refactoring, namely a dedicated Config struct as suggested by Claude Sonnet 4, and also track and hypothesis management classes similar to those proposed by GPT-4o.

Metaprompting The inconsistent nature of some of the LLMs’ refactoring suggestions may also have been caused by my own broad and unspecific prompting, lacking clear requirements. To address this, I subsequently regularly used the technique of metaprompting [64], i.e. leveraging LLMs to refine and structure prompts. This approach is particularly valuable, as LLM response quality has been found to depend significantly on how clearly the objective and its context are specified [65].

One important caveat is worth mentioning: using models from the same family or vendor for both metaprompt generation and prompt execution may introduce bias into the results. Since similar model architectures could inherently be better at interpreting prompts generated by related models, this setup might lead to artificially inflated performance. However, based on my observations, this potential bias did not appear to significantly affect the outcomes.

2.5.2 Documentation Generation for the V0Fitter Refactor³⁷

As an initial test of agentic AI capabilities, I tasked GitHub Copilot with generating Doxygen documentation [66] for the refactored V0Fitter code, using Copilot’s Agent mode in VS Code.

First, I generated a unified set of agent instructions by metaprompting ChatGPT [67]. I then repeatedly invoked the GitHub Copilot agent with the same prompt, each time using a different one of the chosen models, and resetting the repository to a clean state³⁸ before dispatching the request. In addition to the instruction prompt, the agents were provided with the source files to be edited, a “doxygen-cheatsheet”³⁹, and the *Documentation* section from the basf2 coding conventions [68] as context.

However, the information provided might not have been specific enough, seeing as the agents produced inconsistent documentation outputs.

To address this, I reformulated the prompt as a concrete list of requirements, explicitly requiring every class and member to be documented, and directly included the desired Doxygen syntax in the instructions. Additionally, I modified the prompt context to only include the source files of the refactored version, as well as the original V0Fitter’s header file, hoping that the documentation contained within would enhance the model’s comprehension of the code logic.

Reviewing the code generated by the selected models, I made the following observations:

Claude Sonnet 4 The GitHub Copilot agent using this underlying model for generation produced generally meaningful documentation in both prompt iterations, though it often just rephrased variable or method names. Notably, however, it was able to correctly infer ambiguous return values (Lst. 6, 1).

In the second prompt iteration, the added context of the current V0Fitter documentation appears to have enabled the model to infer more about the code logic, such as interpreting the meanings of the different fitter mode values (Lst. 6, 2). Additionally, the documentation generated for newly introduced structures without a counterpart in the original V0Fitter was generally reasonable, but at times too vague to determine whether the underlying logic had actually been understood (Lst. 6, 3).

Gemini 2.5 Pro also generated relatively precise documentation, but, similarly to Claude Sonnet 4, it tended to simply rephrase the names of symbols, and produced non-descriptive comments in some instances. For example, it generated virtually the same comment as Claude Sonnet 4 in Listing 6, (3).

Additionally, Gemini 2.5 Pro occasionally misinterpreted the meaning of values, e.g. the boolean returned by checkInnerHits (Lst. 6, 1) was interpreted as a detection result by the model.

³⁷[github/philipobi-bachelor/chats/v0fitter/documentation-generation-test](https://github.com/philipobi-bachelor/chats/v0fitter/documentation-generation-test)

³⁸Commit b5d0721 in [github/philipobi-bachelor/baf2](https://github.com/philipobi-bachelor/baf2); fitter header file undocumented at this stage

³⁹[github/philipobi-bachelor/chats/v0fitter/documentation-generation-test/initial-prompt/doxygen-overview.md](https://github.com/philipobi-bachelor/chats/v0fitter/documentation-generation-test/initial-prompt/doxygen-overview.md)

Listing 6: Annotated excerpts from the Claude Sonnet 4 agent diffs illustrating notable edits (from both prompt iterations)

```
(1) Correct interpretation of return value - success flag and not detection result:
+    /**
+     * Check for inner hits relative to vertex position.
+     *
+     * @param vertexPos Position of the V0 vertex
+     * @return True if check completed successfully
+     */
    bool checkInnerHits(const ROOT::Math::XYZVector& vertexPos);

(2) Correct inference of v0FitterMode meaning:
+    const int v0FitterMode; ///< V0 fitter mode (0: store at first fit, 1: remove
→ inner hits, 2: mode 1 + SVD restrictions)

(3) pdgClosest = PDG code of the track fit result with mass closest to that of the V0
→ hypothesis:
+    const uint m_pdgClosest = 0; ///< PDG code of closest particle hypothesis
```

On the second prompt iteration, it followed the instructions as specified, during the first prompt however, a friend declaration was modified, and a destructor declaration removed, despite being told not to alter any code logic.

GPT-4o For the first prompt, the documentation generated was somewhat meaningful, however several symbols were incorrectly interpreted, such as the booleans returned by `checkInnerHits` or `checkSharedInnermostCluster`, and the model failed to document several class members altogether.

Processing of the second prompt consistently failed with the error message: “Error: Sorry, the response hit the length limit. Please rephrase your prompt.”

o4-mini Prompting GitHub Copilot with o4-mini as underlying LLM produced some surprisingly insightful documentation, even for the initial prompt that didn’t include any existing documentation as context. Most notably, the non-trivial underlying logic of the `v0FitterMode` values, the `TrackContainer` destructor, as well as the `pdgClosest` variable was partly inferred from the code (Lst. 7). However, the model left all data members undocumented and removed necessary forward declarations in the header file.

In response to the second prompt, the agent no longer modified any code logic, while maintaining more or less the same documentation quality and adding documentation for some, but not all members. That being said, the model seems to have misunderstood the instruction to use inline comments for member documentation, using them universally throughout the code.

In hindsight, I should have explicitly specified in the prompt, how non-member documentation should be handled, as this might have prevented the issue described.

Overall, I found Claude Sonnet 4 to be the most consistent, while o4-mini stood out for its relatively insightful documentation that avoided simply restating declaration names.

2.5.3 V0Fitter Optimization Test⁴⁰

To further test the agentic capabilities of the four tested AI models, I optimistically tasked them with the broad goal of optimizing the V0Fitter’s performance.

As a first step, I generated a unified set of instructions by metaprompting Anthropic’s Claude Sonnet 3.7 Thinking model⁴¹, and set the context to include the original V0Fitter source code. To ensure consistency and comparability of the testing results, for each of the models, I reverted the repository to the release-09-00-00

⁴⁰github.com/philipobi-bachelor/chats/v0fitter/optimization-test

⁴¹A version of Claude Sonnet 3.7 [69] configured to use extended reasoning effort

Listing 7: Excerpts from the o4-mini agent diffs illustrating notable edits in response to the first prompt iteration

```
(1) Inference of the v0FitterMode meaning from the code:
+     * @param v0FitterMode Mode of operation (0=force
+     → store,1=standard,2=useOneSVDHitPair).
Note: v0FitterMode determines the values of the forcestore and useOnlyOneSVDHitPair
      → bool configuration members
(2) The destructor resets the cardinal rep. of the wrapped genfit::Track, if modified
+     /**
+      * @brief Destructor resets smudge on original container if applied.
+      */
     ~TrackContainer();
(3) More meaningful documentation for pdgClosest than other models:
+     /**
+      * @brief Construct TrackContainer wrapping a RecoTrack and PDG hypothesis.
+      * @param recoTrack Pointer to RecoTrack object.
+      * @param pdgClosest PDG code of closest mass hypothesis.
+      * @param isOriginal Flag indicating original track container.
+      */
     TrackContainer(
         RecoTrack* recoTrack,
         const uint pdgClosest,
         const bool isOriginal = true
     );
```

commit before dispatching the prompt via the Copilot Agent mode chat, and then left the agents to complete the task autonomously. I refrained from any further input, except to confirm continuation of execution in case of long runtimes.

While examining the code produced by the different models, I made several observations:

Claude Sonnet 4 made adjustments that may have a positive impact on performance, though not a significant one. The changes primarily target computations that are either inexpensive or likely to be optimized by the compiler anyway, for example, caching the size of a vector for use in a loop bound check, or promoting a vector to a class member that is used to store the pointers of the genfit::Track pair.

On the other hand, a modest improvement to code organization was made by introducing the TrackCache struct, which encapsulates frequently accessed values, such as a pointer to the track fit result with mass closest to that of the hypothesis particle, its corresponding genfit::Track representation and PDG code.

The most significant modification made by the model was transforming the GFRaveVertexFactory object into a reusable class member. This change reduces runtime overhead, as repeated invocations of the encapsulated rave::VertexFactory's wipe method are relatively costly (Fig. 10).

Fitting a vertex using the factory's create method already invokes wipe prior to the actual vertex fit taking place, therefore constructing a new factory object for each fit introduces unnecessary overhead, especially since wipe is called again during object destruction. Based on my observations, the wipe method fully resets the internal state of the vertex factory, making it safe for reuse. For this reason, I have also made the GFRaveVertexFactory a class member in the refactored V0Fitter implementation.

Ultimately, though, the generated code failed to compile, owing to a minor error in the unique_ptr declaration of the GFRaveVertexFactory member.

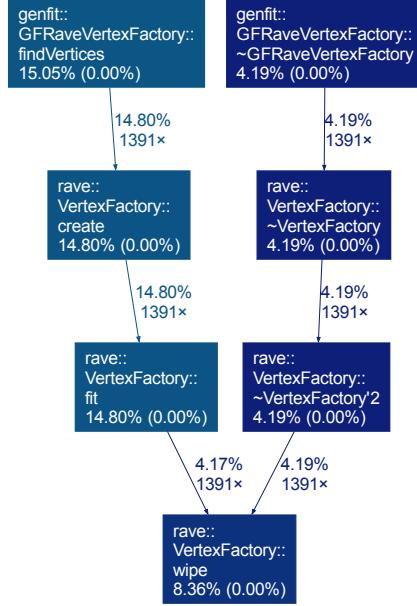


Figure 10: Call graph segment tracing invocations of the `rave::VertexFactory::wipe` method; annotated with profiling metrics

Gemini 2.5 Pro While the GitHub Copilot agent backed by Gemini 2.5 Pro produced code that compiles successfully, it made significant changes to the code logic, despite explicit instructions not to do so.

The hit removal loop boundary conditions were altered in a manner that deviates entirely from the original implementation, indicating a fundamental misunderstanding of the underlying functionality of the code (Lst. 8). Additionally, the agent removed several log outputs from the code and introduced caching of `genfit::Track` retrievals, which unnecessarily avoids calls to the lightweight `RecoTrackGenfitAccess::getGenfitTrack` interface method.

However, it also implemented a meaningful optimization by adjusting the code to only refit `RecoTracks` that contain inner hits, since by default, the `V0Fitter` performs a single refit of tracks without inner hits at the beginning of the hit removal iterations.

Given that the positive impact of this operation on the removal process is unclear and it introduces considerable runtime overhead, both the `NewV0Fitter` and my refactored version of the `V0Fitter` have removed this functionality as well.

GPT-4o made changes that may slightly improve performance, such as adding an `unordered_map` to cache the lookup of daughter particle hypotheses for a given `V0` hypothesis. It is doubtful whether this offers any speed advantage over the original implementation, which uses conditional statements to map the four `V0` hypotheses to their respective daughter particles.

On the other hand, the agent removed multiple log outputs and comments in the code, and more concerningly, removed key parts of the code altogether: Essential logic was deleted without any replacement or explanation, such as the statements setting the track’s cardinal representation before vertex fitting, and those storing `V0` objects —the main purpose of the module (Lst. 9, 1). Additionally, the entire inner hit removal logic was eliminated (Lst. 9, 2), and the parameters used to store the `ValidationV0` object were modified, resulting in a compilation failure.

o4-mini Prompting the GitHub Copilot agent with o4-mini selected as underlying LLM produced only a few, but relatively reasonable changes, and the modified code compiled successfully.

The agent converted the `GFRaveVertexFactory` and `TrackFitter` objects into class members, along with the vector containing the pair of `genfit::Track` pointers that are passed to the vertex factory for fitting. While I am confident that the first modification constitutes a meaningful and computationally safe optimization, I am uncertain about the effects of making the `TrackFitter` object a class member: seeing as it has a `resetFitter`

Listing 8: Gemini 2.5 Pro agent edit that significantly modifies the hit removal iteration limit in fitAndStore, replacing the previous fixed limit of 5 with the total number of hits in the RecoTrack pair

```

unsigned int count_removeInnerHits = 0;
while (hasInnerHitStatus != 0) {
-   ++count_removeInnerHits;
+   /// if the number of trials exceeds the number of hits, terminate the loop.
+   if (count_removeInnerHits > (recoTrackPlus->getRecoHitInformations().size() +
→ recoTrackMinus->getRecoHitInformations().size())) {
+     failflag = true;
+     B2WARNING("V0Fitter: too many trials of removing inner hits.");
+     break;
+   }
+   count_removeInnerHits++;
...
-   if (count_removeInnerHits >= 5) {
-     B2WARNING("Inner hits remained after " << count_removeInnerHits << " times of
→ removing inner hits!");
-     failflag = true;
-     break;
-   }
...
}/// end of the while loop

```

Listing 9: GPT-4o agent edits removing critical V0Fitter functionality

(1) Removal of V0 storage in vertexFitWithRecoTracks:

```

-   B2DEBUG(20, "Creating new V0.");
-   auto v0 = m_v0s.appendNew(std::make_pair(trackPlus, tfrPlusVtx),
-                           std::make_pair(trackMinus, tfrMinusVtx),
-                           posVert.X(), posVert.Y(), posVert.Z());

```

(2) Elimination of inner hit removal logic:

```

+// Optimization in removeInnerHits: Avoid redundant copying and refitting.
bool V0Fitter::removeInnerHits(RecoTrack* prevRecoTrack, RecoTrack* recoTrack,
                               const int trackPDG, const ROOT::Math::XYZVector&
→ vertexPosition)
{
...
-   for (nRemoveHits = 0; nRemoveHits < recoHitInformations.size(); ++nRemoveHits) {
-     if (!prevRecoHitInformations[nRemoveHits]->useInFit()) {
-       recoHitInformations[nRemoveHits]->setUseInFit(false);
-       continue;
-     }
...
-     if (extralength > 0) {
-       recoHitInformations[nRemoveHits]->setUseInFit(false);
-     } else
-       break;
...
-   }
...
-   /// fit recoTrack
-   TrackFitter fitter;
-   if (not fitter.fit(*recoTrack, particleUsedForFitting)) {
...
-   }
   return true;
}

```

method, it may maintain some internal state. Additionally, profiling data indicates that TrackFitter construction and destruction accounts for only about 0.03 % of total instructions, so eliminated overhead is negligible.

Overall, although the AI agents' performance on this optimization task was underwhelming, it was not particularly surprising. It was to be expected that LLMs would face considerable difficulties in generating code that integrates effectively into basf2's extensive codebase and complex logic.

2.5.4 Use of AI during coding

ChatGPT was the AI tool I relied on most during the V0Fitter refactoring, primarily for C++-related questions, but also for queries involving Linux utilities and general development issues. While the value and quality of responses varied, I found it to be very helpful overall.

Listing 10 shows an example of a particularly helpful response I received, while Listing 11 illustrates one of the key limitations of LLMs: their tendency to generate plausible but incorrect output with apparent confidence.

Listing 10: ChatGPT output that helped me recognize reference invalidation due to vector reallocation as the cause of observed segmentation faults⁴²

Prompt: c++ is a reference to a object member invalidated when the member changes?

ChatGPT (o4-mini):

In C++ a reference, once bound, always refers to the very same object (or sub-object) for its entire lifetime. Simply mutating that object (or its member) does **not** “invalidate” the reference. What *can* invalidate a reference (or pointer) is:

1. **End of lifetime**

If the object (or its containing sub-object) is destroyed, any references to it become dangling.

2. **Relocation**

If the object's storage is moved in memory (for example, elements of a `std::vector` being reallocated when you insert past its capacity), then references (and pointers or iterators) to those elements no longer point to the original storage and are therefore invalid.

...

Listing 11: ChatGPT recommending to initialize a `reference_wrapper` by dereferencing a `nullptr`⁴³

Prompt: set static reference member of a class from a function

ChatGPT (o4-mini):

...

1. **Initialize the static reference at its definition** (outside the class).

2. **Use a pointer (or a `std::reference_wrapper`) instead, so you can assign it later.**

...

```
// MyClass.cpp
#include "MyClass.hpp"
```

```
// You still have to "define" it somewhere. This default-constructs the wrapper.
// Internally, reference_wrapper keeps a pointer-but it starts as a null pointer.
```

```
std::reference_wrapper<Foo> MyClass::globalFooRef =
    std::reference_wrapper<Foo>(*(Foo*)nullptr);
```

...

⁴²[github/philipobi-bachelor/chats/v0fitter/questions/chatgpt/C++-Reference-Invalidation.md](https://github.com/philipobi-bachelor/chats/v0fitter/questions/chatgpt/C%2B%2B-Reference-Invalidation.md)

⁴³[github/philipobi-bachelor/chats/v0fitter/questions/chatgpt/Static-reference-initialization.md](https://github.com/philipobi-bachelor/chats/v0fitter/questions/chatgpt/Static-reference-initialization.md)

Copilot Ask Mode and Code Completions During the V0Fitter refactoring process, I occasionally used the Ask mode of GitHub Copilot Chat, primarily querying Claude Sonnet 4 and Gemini 2.5 Pro, as benchmarks indicated these models were best-suited for programming-related tasks (see Sec. 4). However, I disabled Copilot’s automatic code completion feature early on in development, as the suggestions proved largely unhelpful and disruptive to the workflow (cf. Fig. 3, Left).

When responding to questions related to the refactoring work, as well as the broader basf2 codebase, Copilot demonstrated a profound comprehension of the indexed repository structure. It typically answered specific questions accurately when referencing relevant files, or even the entire codebase as context.

For instance, GitHub Copilot successfully identified a bug in my refactored code (Lst. 12).

However, the models sometimes presented incorrect information with high certainty, similar to the reliability issues discussed earlier (Listing 13).

Overall though, Copilot’s Ask mode proved very valuable for codebase exploration, serving as both an intelligent search tool and a documentation system that could synthesize information from multiple files. While useful as an initial resource, I found it necessary to independently verify important technical details to ensure correctness.

Listing 12: GitHub Copilot detects a bug caused by copy-pasting a statement without fully updating its content.⁴⁴

Note: #file is used to add a file to the chat context.

philipobi:

Why does the implementation of Track::checkSharedInnermostCluster in #file:NewV0Fitter.cc produce different results than V0Fitter::checkSharedInnermostCluster in #file:V0Fitter.cc

Variables: file:V0Fitter.cc, file:NewV0Fitter.cc

GitHub Copilot (gemini-2.5-pro):

...

1. **Critical Bug in tracking/v0Finding/fitter/src/NewV0Fitter.cc:** In tracking/v0Finding/fitter/src/NewV0Fitter.cc, there’s a significant bug in how the first hit of the minus track is accessed:

```
// filepath: /project/agkuhr/users/pobi/b2/baf2-v1/tracking/v0Finding/fitter/src/
  ↵ NewV0Fitter.cc
// ...existing code...
const auto& firstHitInfoPlus = *hitInfoSlicePlus[0];
const auto& firstHitInfoMinus = *hitInfoSlicePlus[0]; // BUG: Should be
  ↵ *hitInfoSliceMinus[0]
// ...existing code...
```

...

Listing 13: GitHub Copilot incorrectly suggests that addHitsFromRecoTrack copies the useInFit attribute of RecoHitInformation objects⁴⁵

philipobi:

#file:RecoTrack.h Is the useInFit attribute of the RecoHitInformation objects copied when using addHitsFromRecoTrack?

Variables: file:RecoTrack.h

GitHub Copilot (claude-sonnet-4):

Looking at the addHitsFromRecoTrack method and the addHit template method in the RecoTrack class, I can see that yes, the useInFit attribute is copied when using addHitsFromRecoTrack. Here’s why:

...

⁴⁴[github/philipobi-bachelor/chats/v0fitter/questions/copilot/bug-find.md](https://github.com/philipobi-bachelor/chats/v0fitter/questions/copilot/bug-find.md)

⁴⁵[github/philipobi-bachelor/chats/v0fitter/questions/copilot/useInFit-copy.md](https://github.com/philipobi-bachelor/chats/v0fitter/questions/copilot/useInFit-copy.md)

2.6 Additional Tools and Methods

This section provides a brief overview of further noteworthy tools and methodology I used throughout the refactoring process and in working with basf2 overall.

2.6.1 OverlayFS

OverlayFS [70] is a Linux kernel file system that allows multiple directory layers to be combined into a single, unified view. It achieves this by overlaying an upper directory (read-write) on top of a lower directory (typically read-only), presenting both layers as one merged file system at a specified mount point (see Fig. 11 for a visualization).

When a file from the lower layer is modified, OverlayFS uses a copy-up mechanism: the file is first copied to the upper layer, and changes are applied there. This ensures that the lower layer remains unaltered, while still allowing full read-write access through the upper layer.

During the V0Fitter refactoring process, I used OverlayFS to create a safe and flexible development environment by setting up overlay mounts for the basf2 repository and the software externals. This approach was especially helpful in the early stages of my work, when I was still getting familiar with the framework and often introduced breaking changes. With OverlayFS, I could easily return to a clean state by discarding the upper layer, significantly reducing the time required for environment recovery.

However, due to limited privileges on the shared Linux server used for development, I was restricted to using fuse-overlayfs [71], a user-space implementation with reduced performance compared to the kernel-mode OverlayFS.

Interestingly, I first learned about OverlayFS during a conversation with ChatGPT, when it recommended OverlayFS in response to my query for a method to efficiently mirror a directory in Linux in such a way that the original remained untouched.⁴⁶

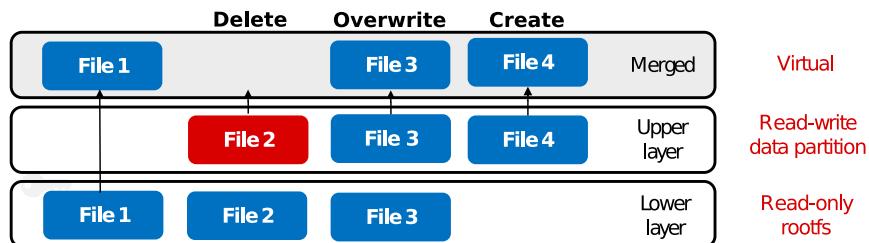


Figure 11: Schematic: OverlayFS layered architecture and impact of file operations on directory layers [72]

2.6.2 VS Code GDB Debugging

To decouple the UI from debugging backends, VS Code uses Debug Adapters that communicate with the UI via Microsoft's open-source Debug Adapter Protocol (DAP) [73]. The editor's C/C++ extension [74] implements such adapters for multiple debuggers, notably the GNU debugger (GDB) [75] on Linux.

Debugging a C/C++ binary involves defining a configuration in a launch.json file, where options such as setup commands for the debugger, program arguments, and environment variables can be specified.⁴⁷ These setup commands for example allow to enable GDB's pretty-printing capabilities for C++ STL containers, resulting in improved display of objects like std::vectors in the debugging UI (see Fig. 12, Left).

VS Code's debugging interface offers helpful features such as navigating between call stack frames and inspecting their variables in a side panel, while displaying the current execution position within the selected stack frame in the corresponding source file. Additionally, both standard and conditional breakpoints can be set either through a dedicated panel or by clicking the area next to the source line number in the editor (Fig. 12).

⁴⁶github.com/philipobi-bachelor/chats/v0fitter/questions/chatgpt/OverlayFS-Mirror-Setup.md

⁴⁷For the launch.json file I used for basf2 debugging, see github.com/philipobi-bachelor/project/artifacts/debugging.md

The screenshot shows the VS Code interface during a remote debugging session of the basf2-v1 project. The left side of the interface is a GDB debugger panel with tabs for RUN AND DEBUG, VARIABLES, WATCH, CALL STACK, and BREAKPOINTS. The VARIABLES tab shows a list of local variables, including hitInfos, m_flag, m_useInFit, and m_createdTrackPointID. The CALL STACK tab shows the call stack from libtracking.so!Belle2::NewV0Fitter::Track::removeInnerHits() up to libframework.so!Belle2::EventProcessor::processEvent(). The BREAKPOINTS tab lists two breakpoints: V0FinderModule::event (hit) and NewV0Fitter.cc tracking/v0Finding/fitter/src (hit). The right side of the interface is an editor window for the file NewV0Fitter.cc. Line 411 is highlighted with a yellow background, and several other lines in the code are also highlighted with yellow, indicating they are breakpoint locations. The code itself is in C++ and defines a namespace Belle2 and a class NewV0Fitter.

Figure 12: Remote debugging of basf2 with VS Code and GDB

Left: Debugging panel with variable, call stack and breakpoint inspector

Right: Editor view with highlighted execution position and display of breakpoint locations

2.6.3 V0Fitter debugging

Using VS Codes debugging features, I was able to efficiently debug different V0Fitter implementations side-by-side, comparing their event handling. To identify and analyze differences, I introduced event and track pair counters along with debug print statements into the module and fitter code.

Using vimdiff [76] to compare the log outputs from running v0ValidationGenerateSample with the different fitter versions then allowed me to pinpoint events with divergent behavior. By subsequently setting conditional breakpoints using the counter variables' values, I could pause execution in both debugging sessions at the relevant positions, and step through processing in parallel. It was through this process that I discovered that the primary reason for the NewV0Fitter's speed advantage was its strict invariant mass cut, which led to fewer track refitting calls.

VS Code's debugging capabilities also proved valuable for general debugging purposes during the development process.

2.6.4 Clangd Extension

The clangd extension for VS Code [77] implements a client that communicates with clangd [78], a language server for C, C++ and Objective-C built on top of the Clang / LLVM compiler infrastructure. By utilizing clangd's parsing, indexing and error analysis capabilities, the extension offers IDE-level functionality such as:

- Navigation to symbol definitions across the entire codebase
- Inline display of compilation errors, warnings and static analysis results
- Display of type information, documentation and function signatures when hovering over symbols
- Project-wide symbol renaming functionality
- Auto-completion of symbol names and semantic code highlighting
- Inline annotation of types and function parameters (see gray code annot. in Fig. 12, Right and Fig. 3, Left)

While the official VS Code C/C++ extension provides completion and analysis functionality as well, I found

its engine to be less robust and performant, particularly for large projects, therefore I chose to disable this functionality in favor of clangd, while still using its debugging features as mentioned above.

To accurately parse all components of a project, clangd depends on a JSON compilation database, typically called `compile_commands.json`. Most build systems can create these databases automatically during the build, for basf2 this is achieved by compiling with `scons --create-json <additional options>`.

2.6.5 Chat Exporting

To document and compare GitHub Copilot’s performance across different tasks, prompt formulations, and underlying LLMs, I needed a way to export AI interaction logs that were human-readable and information-rich.

The Copilot extension in VS Code provides limited export functionality, supporting only JSON format or manual copying of chat content in Markdown format. Additionally, neither export method preserves model information, making later comparisons impossible.

To solve this problem, I developed a custom chat archiving utility that converts JSON chat logs exported from VS Code into annotated Markdown files. The tool adds important metadata to the exported logs, especially the underlying model used for each response or code edit. It also automatically generates diffs for any file changes made during a conversation. All excerpts of Copilot chat logs and edit diffs shown in this section are from logs files generated using this chat archiver. A brief implementation overview of the archiving utility is provided in Section 5.

For exporting ChatGPT conversations, I used ChatGPT Exporter [79], a browser script that supports single or bulk chat log exports in multiple formats. I used its Markdown export feature to save notable ChatGPT conversations for documentation purposes.

3 Generic Code Optimization

To further evaluate GitHub Copilot’s capabilities in software optimization tasks, I followed my thesis advisor Dr. Lück’s suggestion to test its performance on generic code examples containing common C++ inefficiencies.

3.1 Methodology

Input Data As a first step, I produced a set of such examples by prompting LLMs to generate code with typical C++ performance mistakes. Initially, I requested Gemini 2.5 Pro to produce 100 examples: For each example I required a standalone C++ code snippet⁴⁸ containing an inefficient operation, a corresponding optimized version performing the same task, a brief title and a description of the inefficiency.

However, initial evaluation revealed quality issues in many examples. While the concepts were generally sound, code snippets often lacked proper implementation. For example, function bodies were frequently left empty or contained only trivial operations with no side effects, preventing the intended inefficiencies from actually affecting the execution performance (see Lst. 14). Additionally, several examples were nearly identical.

To address these issues, I refined the prompt to require each example to output a result value, ensuring observable side effects. I also reduced the number of requested examples to 50 and explicitly required distinctness. This revised prompt was submitted to Claude Sonnet 3.7 Thinking and produced higher quality examples overall, though some similarities between individual examples remained.

I manually curated the final dataset by replacing several examples from the second iteration with high-quality ones from the initial set. The result was a Markdown file containing 50 numbered examples, each with inefficient and optimized code blocks, a title and a description.⁴⁹ I then programmatically extracted these examples using regular expressions⁵⁰ in Python, and stored them in a database.

Test File Generation To create realistic test conditions, I generated two complementary test files for each example using Python. By sampling a Bernoulli random variable with parameter 0.5 and a predetermined random seed, I randomly assigned implementations to files. A value of 1 placed the efficient version in the first test file and the inefficient version in the second.

This approach produced two test files, each containing 50 tasks with mixed efficient and inefficient code. I employed this methodology because it better reflects real-world scenarios, where codebases typically contain a mixture of efficient and inefficient implementations rather than being uniformly optimized or unoptimized.

Evaluation Process I evaluated GitHub Copilot’s performance across the different underlying models using the prepared test files. For each model and test file combination, I tasked Copilot with improving the code. The instruction prompt, generated using Claude 3.7 Sonnet Thinking, avoided revealing the deliberate test nature. Instead, it presented each file as “a C++ file that performs various computational tasks”. The instructions required that any modifications (1) leave functionality unchanged, and (2) provide significant performance improvements to prevent superficial changes without meaningful benefits.

After each agent completed its optimization task, I archived the conversations and file modifications using my chat archiving utility, and reset the test files to their original unmodified state. From the file edit information in the exported chat logs, I extracted the modified code implementation of each task, and conducted a manual high-level review comparing each example’s post-optimization code against its original version.

This evaluation process allowed me to assess how many inefficient examples Copilot successfully identified and attempted to optimize, as well as whether Copilot made additional improvements to examples that were already efficiently implemented.

Benchmarking To complement the qualitative analysis, I performed comprehensive benchmarking of all code variants, measuring the performance of original efficient versions, original inefficient versions, and Copilot-

⁴⁸I.e. complete with includes and main function

⁴⁹[github/philipobi-bachelor/code-optimization-test/examples.md](https://github.com/philipobi-bachelor/code-optimization-test/examples.md)

⁵⁰**Note:** for working with regular expressions (regex) [80], I found the website Regex101 [81] invaluable. Key features include:

- Interactive regex editing with real-time syntax highlighting, pattern explanations, and match visualization
- Quick reference materials
- Code generation for regex processing in multiple programming languages
- Regex debugging functionality

modified versions. This approach quantified performance differences between the reference versions and measured Copilot’s optimization impact. The process also verified that examples compiled successfully and executed without errors.

For consistency and reproducibility, all benchmarking was conducted in a containerized environment, using a custom image that links code with the nanobench library [82] during compilation and then executes the binary. Benchmarking is automated through a Python script that instruments code snippets with nanobench facilities and launches the container, transmitting instrumented code via standard input. Results and error messages are captured from the container’s standard error stream⁵¹ and processed by the script.

All benchmarking data is stored in a dedicated database for analysis and documentation purposes.

Note: Detailed documentation for this test is available in a dedicated repository.⁵² This includes the Python scripts used for data extraction, benchmarking and evaluation, as well as the build specification for the benchmarking container image. The repository also contains the metaprompts used to generate the examples and agent instructions, and complete chat logs with exact prompts, model responses and code edits.

The extracted examples and benchmark data are preserved in a separate repository containing backups of the central database used throughout this thesis.⁵³ This database also stores the raw JSON data from every chat log archived during the research process.

3.2 Results

From the collected evaluation and benchmarking data, I programmatically generated comprehensive results tables that present relevant information regarding GitHub Copilot’s optimization performance across the different test scenarios.

Tables 7 and 8 present relevant result data for test files 1 and 2 respectively when compiled without optimizations (-O0 flag). Tables 14 and 15 in the appendix show the corresponding results when the code was compiled with compiler optimizations enabled (-O3 flag).

Table 6 summarizes the optimization test results, showing the distribution of inefficient implementations between the two test files, and the number of examples successfully improved by each GitHub Copilot model.

	Improved Examples		
	Test 1	Test 2	Sum
Ineff. Example Count	22	28	50
Claude Sonnet 4	21	20	41
Gemini 2.5 Pro	15	22	37
GPT-4o	2	7	9
o4-mini	11	15	26

Table 6: Summary of optimization improvements by model across test files

Legend for Tables 7, 8, 14, 15

- **Task number / Description:** Short description of the inefficiency in each test example; full implementation details available in the Markdown file containing all examples selected for testing⁵⁵
- **average(elapsed):** Mean wall-clock execution time measured by nanobench for one execution of the example code
- **bl:** Baseline performance representing *average(elapsed)* for the efficient implementation of each task

⁵¹As the standard output is mostly cluttered with example code output

⁵²github.com/philipobi-bachelor/code-optimization-test

⁵³github.com/philipobi-bachelor/project-db-dump

⁵⁴github.com/philipobi-bachelor/code-optimization-test/chat-logs/generated-examples-1.md

⁵⁵github.com/philipobi-bachelor/code-optimization-test/examples.md

Listing 14: Example generated by Gemini 2.5 Pro illustrating the issue of missing function implementations⁵⁴

philipobi:

Your task is to produce **100** standalone C++ code examples that are intentionally inefficient. Each example should be used as an exercise for students to **find and fix** the inefficiency.

...

GitHub Copilot (gemini-2.5-pro):

Of course. Here are 100 intentionally inefficient C++ code examples for students to find and fix.

...

15. **Title:** Unnecessary `std::string` creation from `const char*`

```
#include <iostream>
#include <string>
#include <map>
void process_string(const std::string& s) {
    // Does something with the string
}
int main() {
    process_string("this is a string literal");
    return 0;
}
```

Fix: Creating a `std::string` from a string literal involves memory allocation. Overload the function to accept `const char*` or use `std::string_view` (C++17) to avoid allocation.

```
#include <iostream>
#include <string>
#include <string_view>
void process_string(std::string_view s) {
    // Does something with the string_view
}
int main() {
    process_string("this is a string literal");
    return 0;
}
...
```

- **s/bl:** Performance ratio $\log_{10}(slow/bl)$, where *slow* is *average(elapsed)* for the inefficient implementation; a measure of performance degradation of the inefficient version relative to the efficient baseline
- **s?(=slow?):** Indicator denoting which implementation version was in the test file; filled circle: inefficient version present; empty circle: efficient version present
- **Model columns:** Each model's results are displayed in two columns:
 - **Left column:** Modification status indicator
 - * Filled circle: Inefficient implementation was present and Copilot modified the code
 - * Half-filled circle: Efficient implementation was present and Copilot modified the code
 - * Empty circle: Copilot made no modifications to the code
 - * Red highlighting: Inefficient implementation was present but Copilot failed to modify the code
 - * Green highlighting: Inefficient implementation was present and Copilot modified the code
 - **Right column:** Performance metric
 - * Value equals $\log_{10}(c/bl)$, where *c* is *average(elapsed)* for the example's code after Copilot's modifications
 - * No highlighting: Efficient version was present and $\log_{10}(c/bl) \in (-0.1, 0.1)$
 - * Color mapping for other cases: Green tones denote performance similar or better than baseline; yellow and red tones show performance degradation
- **Model abbreviations:** c-s-4 = Claude Sonnet 4, g-2.5-p = Gemini 2.5 Pro

- **Note:** I implemented cell coloring to enhance the interpretability of model performance across the large dataset by highlighting patterns that may be difficult to detect from the raw data alone.

Notable Observations

The evaluation results demonstrate clear performance differences between the underlying models powering GitHub Copilot.

Claude Sonnet 4 emerged as the strongest performer, successfully detecting and optimizing the highest number of inefficient examples across both test files (Tab. 6).

This superior performance is further corroborated by the detailed benchmarking data, which shows that code implementations modified by the Claude Sonnet-powered Copilot typically achieved at least baseline performance levels, with multiple cases even demonstrating improvements beyond the baseline efficient versions (Tab. 7, 8).

Example 43 in test file 1 illustrates a case where Claude Sonnet 4 successfully identified an optimization opportunity but did not achieve sufficient improvement to reach baseline performance (Tab. 7). The inefficient version employs a `std::map` for storing and retrieving values, using dense integer keys ranging from 0 to 9999. The efficient reference implementation replaces this structure with a `std::vector`, utilizing the integer key directly as a vector index for value access.

Claude Sonnet 4's optimization involved replacing the `std::map` with a `std::unordered_map` to enable faster key access, while additionally performing accumulation of the stored values directly during insertion, instead of looping over the keys twice. Although benchmarking data confirms this modification improves performance, the implementation still exhibits considerable performance overhead compared to the vector-based approach.

Example 34 demonstrates a more sophisticated optimization where Claude Sonnet 4 effectively “outsmarted” the test design. This example involves repeated string lookups in a set, with the inefficient reference implementation using `std::set` and the efficient version employing `std::unordered_set` for constant-time lookup complexity.

In both test scenarios, with either the inefficient version or the efficient reference version present, Claude Sonnet 4 recognized that the search target remained constant across all iterations. Rather than optimizing the data structure, it restructured the code to perform the search operation only once and multiply the result by the number of iterations. This approach yielded dramatic performance improvements, reducing runtime by more than four orders of magnitude compared to the efficient baseline.

Similar “outsmarting” behavior occurred in most cases where significant runtime improvements beyond the baseline were achieved, not exclusively with Claude Sonnet 4 but also with other models, particularly o4-mini and Gemini 2.5 Pro. For instance, in example 34, o4-mini implemented the identical optimization as Claude

Task Number / Description	bl	s/bl	s?	c-s-4	g-2.5-p	gpt-4o	o4-mini
1 Ineff. string concat. in loop	4 ms	0.8	○	● -1.0	● -0.1	● -0.1	○ 0.0
2 Reserving vector capacity	860 μ s	0.1	○	● -0.3	○ 0.0	○ 0.0	○ 0.0
3 Excessive stream flushing	560 μ s	1.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
4 Passing/returning vector by value	78 μ s	0.0	○	● -0.1	● 0.0	○ 0.0	● 0.0
5 Repeated lookups with std::find	2 ms	0.6	●	● 0.0	● 0.0	● 0.0	● -0.9
6 Passing shared_ptr by value	16 ms	0.4	○	● -0.8	○ 0.0	○ 0.0	○ 0.0
7 Temporary objects and emplace_back	660 μ s	0.0	●	● -0.1	● 0.0	○ 0.0	● -0.1
8 Duplicate map lookups	8 ms	0.3	○	● -2.6	● -1.1	○ 0.0	● -2.6
9 Bubble sort vs. std::sort	532 μ s	3.3	●	● 0.0	● 0.0	○ 3.3	● 0.0
10 Reading file one char at a time	140 μ s	0.9	○	○ 0.0	● 0.0	○ 0.0	○ 0.0
11 pow for integer powers of 2	716 ns	0.2	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
12 Repeated calls to vector::erase	146 μ s	1.0	○	● -0.1	● 0.0	○ 0.0	○ 0.0
13 Ineff. str to int conversion	1 μ s	0.2	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
14 std::list for random access	116 μ s	2.1	○	● -0.2	○ 0.0	○ 0.0	○ 0.0
15 Passing large objects by value	2 ms	0.0	○	○ 0.0	○ 0.0	● 0.0	○ 0.0
16 High contention on atomic	3 ms	1.9	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
17 Using std::map when order not needed	28 ms	0.6	●	● -0.1	● 0.0	○ 0.6	○ 0.6
18 Using temp. string for comparison	15 ms	0.4	○	● -1.8	○ 0.0	○ 0.0	○ 0.0
19 Inefficient map iteration	8 ms	1.3	○	● -3.3	○ 0.0	○ 0.0	● -3.2
20 Iteration with auto vs. auto&	8 ms	0.6	○	● -3.8	○ 0.0	○ 0.0	○ 0.0
21 Opening/closing file in loop	83 μ s	2.0	●	● 0.0	● 0.0	○ 2.0	● 0.0
22 std::accumulate with ineff. operation	46 μ s	1.3	○	○ 0.0	● 0.0	○ 0.1	○ 0.0
23 Sorting large list in place	31 ms	0.4	○	○ 0.0	● 0.4	○ 0.0	○ 0.0
24 Loop vs. std::accumulate	7 ms	0.2	●	● 0.0	● 0.0	○ 0.2	○ 0.2
25 Ineff. string concat. in loop	8 ms	1.3	○	○ 0.0	● -0.2	○ 0.0	○ 0.0
26 Virtual function in loop	179 ms	0.0	●	● 0.0	● 0.0	○ 0.0	○ 0.0
27 shared_ptr for owned member	141 μ s	0.1	●	○ 0.1	○ 0.1	○ 0.1	● 0.1
28 Temporary objects and emplace_back	5 ms	0.2	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
29 Re-calculating value in loop	2 ms	0.5	●	● -3.3	○ 0.4	○ 0.5	● -3.3
30 std::vector for fixed-size array	12 ms	0.1	●	● -4.1	● -0.9	○ 0.1	● -4.0
31 Manual loop to fill container	5 ms	0.3	○	● 0.1	○ 0.0	○ 0.0	○ 0.0
32 Repeated object creation in loop	2 ms	0.9	●	● -3.3	○ 0.9	○ 0.9	○ 0.9
33 Repeated stream setup in loop	37 ms	0.0	○	● -4.5	○ 0.0	○ 0.0	○ 0.0
34 std::set for simple existence checks	100 ms	0.4	●	● -4.5	● 0.0	○ 0.4	● -4.5
35 Ineff. string splitting	12 ms	-0.3	●	● -3.7	● -0.4	○ -0.3	○ -0.3
36 Manual vs. range container init.	3 ms	0.8	●	● -0.9	○ 0.8	○ 0.8	○ 0.8
37 Returning large object by value	2 ms	0.6	○	○ 0.0	● 0.6	○ 0.0	○ 0.0
38 std::map for int keys	8 ms	0.3	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
39 Redundant loop bounds check	124 μ s	0.0	●	● -0.2	● 0.0	○ 0.0	○ 0.0
40 Div. and mod. for powers of 2	3 ms	0.0	●	● 0.0	○ 0.0	○ 0.0	○ 0.0
41 std::map for enum keys	1 μ s	0.4	●	● 0.4	○ 0.4	○ 0.4	○ 0.4
42 Ineff. string filtering	31 μ s	1.7	●	● 0.0	● 0.0	○ 1.7	● -0.2
43 map vs. vector for dense int keys	134 μ s	1.8	●	● 1.2	● 1.3	○ 1.9	○ 1.9
44 Column-major matrix traversal	1 s	0.4	●	● 0.0	● 0.0	● 0.0	● 0.0
45 std::function overhead	202 ms	0.3	○	● 0.1	● -0.2	○ 0.0	○ 0.0
46 Modifying front of vector in loop	95 μ s	1.5	●	● 0.4	● 0.0	○ 1.5	● 0.1
47 Linear search on sorted data	95 μ s	0.1	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
48 Recompiling regex repeatedly	111 μ s	0.5	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
49 std::any when type is known	14 μ s	0.9	●	● 0.0	○ 0.9	○ 0.9	○ 0.9
50 Fibonacci without memoization	2 μ s	5.7	○	● -0.2	● -0.2	○ 0.0	○ 0.0

Table 7: Copilot optimization results for Test File 1 (benchmarks compiled with -O0)

Task Number / Description	bl	s/bl	s?	c-s-4	g-2.5-p	gpt-4o	o4-mini
1 Ineff. string concat. in loop	4 ms	0.8	●	● -0.9	● -0.9	● -0.9	● -0.2
2 Reserving vector capacity	860 μ s	0.1	●	● 0.0	● 0.0	● 0.0	● 0.0
3 Excessive stream flushing	560 μ s	1.0	●	○ 1.0	● 0.0	○ 1.0	○ 1.0
4 Passing/returning vector by value	78 μ s	0.0	●	○ 0.0	● 0.0	○ 0.0	○ 0.0
5 Repeated lookups with std::find	2 ms	0.6	○	○ -0.1	○ 0.0	○ 0.0	○ 0.0
6 Passing shared_ptr by value	16 ms	0.4	●	● -0.7	○ 0.4	○ 0.4	○ 0.4
7 Temporary objects and emplace_back	660 μ s	0.0	○	○ -0.1	○ -0.1	○ 0.0	○ 0.0
8 Duplicate map lookups	8 ms	0.3	●	● -2.6	● 0.0	○ 0.3	○ 0.3
9 Bubble sort vs. std::sort	532 μ s	3.3	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
10 Reading file one char at a time	140 μ s	0.9	●	● 0.0	● 0.0	○ 0.8	● 0.0
11 pow for integer powers of 2	716 ns	0.2	●	● 0.0	● 0.0	○ 0.2	○ 0.2
12 Repeated calls to vector::erase	146 μ s	1.0	●	● 0.0	● 0.0	● 0.0	● 0.0
13 Ineff. str to int conversion	1 μ s	0.2	●	● 0.0	● 0.0	○ 0.2	○ 0.2
14 std::list for random access	116 μ s	2.1	●	● 0.0	● 0.0	○ 2.1	○ 2.1
15 Passing large objects by value	2 ms	0.0	●	● 0.0	● 0.0	● -0.4	● 0.0
16 High contention on atomic	3 ms	1.9	●	○ 1.8	○ 1.8	● 1.8	○ 1.8
17 Using std::map when order not needed	28 ms	0.6	○	○ 0.0	○ -0.1	○ 0.0	○ 0.0
18 Using temp. string for comparison	15 ms	0.4	●	● 0.0	● 0.0	○ 0.4	● 0.0
19 Inefficient map iteration	8 ms	1.3	●	● -3.2	● -1.0	○ 1.3	○ 1.3
20 Iteration with auto vs. auto&	8 ms	0.6	●	● 0.0	● -3.8	○ 0.6	● 0.0
21 Opening/closing file in loop	83 μ s	2.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
22 std::accumulate with ineff. operation	46 μ s	1.3	●	○ 1.4	● 0.1	○ 1.3	○ 1.3
23 Sorting large list in place	31 ms	0.4	●	○ 0.4	● -0.2	○ 0.4	○ 0.4
24 Loop vs. std::accumulate	7 ms	0.2	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
25 Ineff. string concat. in loop	8 ms	1.3	●	● -3.5	● -3.5	● 0.0	● -3.5
26 Virtual function in loop	179 ms	0.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
27 shared_ptr for owned member	141 μ s	0.1	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
28 Temporary objects and emplace_back	5 ms	0.2	●	● 0.0	● 0.0	○ 0.2	○ 0.0
29 Re-calculating value in loop	2 ms	0.5	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
30 std::vector for fixed-size array	12 ms	0.1	○	○ -4.2	○ -0.9	○ 0.0	○ 0.0
31 Manual loop to fill container	5 ms	0.3	●	● 0.1	○ 0.3	○ 0.3	● 0.1
32 Repeated object creation in loop	2 ms	0.9	○	○ -1.1	○ 0.0	○ 0.0	○ 0.0
33 Repeated stream setup in loop	37 ms	0.0	●	● -2.4	● -2.4	○ 0.0	● -4.5
34 std::set for simple existence checks	100 ms	0.4	○	○ -4.5	○ 0.0	○ 0.0	○ 0.0
35 Ineff. string splitting	12 ms	-0.3	○	○ -2.8	○ -0.3	○ 0.0	○ 0.0
36 Manual vs. range container init.	3 ms	0.8	○	○ -0.9	○ 0.0	○ 0.0	○ 0.0
37 Returning large object by value	2 ms	0.6	●	○ 0.6	○ 0.6	○ 0.6	● 0.0
38 std::map for int keys	8 ms	0.3	●	○ 0.3	○ 0.3	○ 0.3	○ 0.3
39 Redundant loop bounds check	124 μ s	0.0	○	○ -0.3	○ -0.2	○ 0.0	○ 0.0
40 Div. and mod. for powers of 2	3 ms	0.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
41 std::map for enum keys	1 μ s	0.4	○	○ 0.1	○ -0.1	○ 0.0	○ 0.0
42 Ineff. string filtering	31 μ s	1.7	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
43 map vs. vector for dense int keys	134 μ s	1.8	○	○ -0.2	○ -0.1	○ 0.0	○ 0.0
44 Column-major matrix traversal	1 s	0.4	○	○ -0.8	○ -2.1	○ 0.0	○ 0.0
45 std::function overhead	202 ms	0.3	●	● 0.0	○ 0.3	○ 0.3	● 0.0
46 Modifying front of vector in loop	95 μ s	1.5	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
47 Linear search on sorted data	95 μ s	0.1	●	● 0.0	● -0.1	○ 0.1	● 0.0
48 Recompiling regex repeatedly	111 μ s	0.5	●	● 0.0	● 0.0	○ 0.6	● 0.0
49 std::any when type is known	14 μ s	0.9	○	○ -0.1	○ 0.0	○ 0.0	○ 0.0
50 Fibonacci without memoization	2 μ s	5.7	●	○ -0.3	● -0.2	● -0.2	● -0.2

Table 8: Copilot optimization results for Test File 2 (benchmarks compiled with -O0)

Sonnet 4 when the inefficient version was present, though notably failed to achieve further improvements when the reference efficient version was included in the test file (cf. Lst. 15, Tab. 7, 8).

One exception in the evaluation data concerns example 50 in test file 2 for Claude Sonnet 4. While the model successfully modified and optimized the inefficient example, it did so by significantly altering the code's functionality (Lst. 16). Consequently, I classified this as "not fixing the code" despite benchmarking results showing performance gains from the optimization (Tab. 8).

The results suggest that more common inefficiencies are easier for models to detect and improve. For example, the performance issue introduced by column-major matrix traversal (Example 44, Tab. 7) was successfully identified and optimized by all tested models, even GPT-4o.

Based on the count of detected and improved inefficient examples, as well as a qualitative review of the comprehensive results tables, Gemini 2.5 Pro ranks second in performance, o4-mini third, and GPT-4o last by a wide margin, having detected only 9 out of 50 inefficient examples.

Comparing benchmarking data between -O0 and -O3 compilation flags reveals that while absolute runtimes differ substantially, the performance ratios between inefficient and efficient versions do not show any drastic changes: Many slow examples maintain logarithmic ratios near zero compared to the fast baseline across both compilation settings.

Several factors may explain these observations:

- Compilation with -O0 disables some but not all optimizations
- The primary performance overhead may stem from benchmarking instrumentation or print statements, making the effect of the intentionally introduced inefficiencies insignificant in comparison
- The overall quality of examples is limited, as they were not hand-crafted to maximize performance differences between inefficient and efficient versions

Despite these limitations, I consider the data collected in this evaluation to provide a meaningful assessment of the selected AI models' code optimization capabilities.

Listing 15: "Outsmarting" behavior displayed by o4-mini⁵⁶, (analogous for Claude Sonnet 4)

```
// Task 34
{
+   using namespace std;
    std::set<std::string> words = {"apple", "banana", "orange", "grape", "pear"};
    std::string target = "orange";
-   int count = 0;
-   for (int i = 0; i < 1000000; i++) {
-     if (words.find(target) != words.end()) {
-       count++;
-     }
-   }
+   // cache find result to avoid repeated tree lookups
+   bool exists = (words.find(target) != words.end());
+   int count = exists ? 1000000 : 0;
    std::cout << "Count: " << count << std::endl;
}
```

⁵⁶[github.com/philipobi-bachelor/code-optimization-test/chat-logs/test1/o4-mini.md](https://github.com/philipobi-bachelor/code-optimization-test/blob/main/chat-logs/test1/o4-mini.md)

⁵⁷[github.com/philipobi-bachelor/code-optimization-test/chat-logs/test2/clause-sonnet-4.md](https://github.com/philipobi-bachelor/code-optimization-test/blob/main/chat-logs/test2/clause-sonnet-4.md)

Listing 16: Claude Sonnet 4 modification restricts function to $n \leq 40$ and indiscriminately computes the first 41 Fibonacci numbers⁵⁷

```

namespace task50 {
    // Optimization: Memoized fibonacci to avoid exponential time complexity
    unsigned long long fibonacci(unsigned int n) {
        if (n <= 1)
            return n;
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
    static std::vector<unsigned long long> memo(41, 0);
    static bool initialized = false;
    if (!initialized) {
        memo[0] = 0;
        memo[1] = 1;
        for (int i = 2; i <= 40; ++i) {
            memo[i] = memo[i-1] + memo[i-2];
        }
        initialized = true;
    }
    return memo[n];
}
} // namespace task50
...
// Task 50
{
    using namespace task50;
    unsigned int n = 40;
    unsigned long long result = fibonacci(n);
    std::cout << "Fibonacci(" << n << ") = " << result << std::endl;
}

```

4 Model Selection

4.1 Selection of Models for Comparison

I selected four models from the GitHub Copilot Pro plan, representing the most capable offerings from each major model family: Anthropic’s Claude Sonnet 4, Google Gemini 2.5 Pro, OpenAI’s GPT-4o and OpenAI o4-mini.

My selection was guided by data from two established LLM benchmarks: LMArena [83] (formerly Chatbot Arena [84]) and the Artificial Analysis Index [85]. Based on the benchmarking data, I chose the models that were available both in Ask and Agent modes⁵⁸ and had the highest coding scores (cf. Tab. 9, 10, 11).

However, I made an error in interpreting the benchmark data when selecting a model from OpenAI’s GPT family. Based on LMArena data showing GPT-4o outperforming GPT-4.1 in coding tasks (cf. Tab. 12), I chose GPT-4o for comparison. I later discovered that the high-performing GPT-4o model in benchmarks represents a newer version powering ChatGPT (chatgpt-4o-latest-20250326 in Tab. 12), while Copilot uses an older version (gpt-4o-2024-11-20⁵⁹).

Although I could not find this exact model in LMArena’s data, other GPT-4o versions demonstrated considerably lower performance across benchmark categories (Tab. 10). Similarly, Artificial Analysis, which had evaluated this specific model, found significantly lower performance compared to both GPT-4.1 and the other capable models available in Copilot (Tab. 11).

This version mismatch likely explains the poor performance of GPT-4o-powered Copilot that I observed in my tests. While this oversight was unfortunate, it demonstrates that benchmark performance can indeed be indicative of real-world performance to some extent.

⁵⁸**Note:** during testing, o4-mini was still available for use in Agent mode; now however, this seems to have changed (cf. Tab. 9)

⁵⁹According to Copilot API data github.com/philipobi-bachelor/project/artifacts/models/copilot-models.json

Model		Mode Support			Plan Availability			Request Multiplier	
Vendor / Family	Version	Ask	Edit	Agent	Free	Pro	Pro+	Free	Paid
OpenAI GPT	4.1 [86]	●	●	●	●	●	●	1.00	0.00
	4o [63]	●	●	●	●	●	●	1.00	0.00
OpenAI	o3 [87]	●	●	○	○	○	●	-	1.00
	o4-mini [62]	●	●	○	○	●	●	-	0.33
Anthropic Claude	Opus 4 [88]	●	●	○	○	○	●	-	10.00
	Sonnet 4 [60]	●	●	●	○	●	●	-	1.00
	Sonnet 3.7 Th. [69]	●	●	○	○	●	●	-	1.25
	Sonnet 3.7 [69]	●	●	●	○	●	●	-	1.00
	Sonnet 3.5 [89]	●	●	●	●	●	●	1.00	1.00
Google Gemini	2.5 Pro [61]	●	●	●	○	●	●	-	1.00
	2.0 Flash [90]	●	●	○	●	●	●	1.00	0.25

Table 9: Information on GitHub Copilot model availability (retrieved 2025-08-05 [91])

Model	Hard						Instruction Following	Longer Query
	Overall	Prompts	Coding	Math				
gemini-2.5-pro	1	1	1	1	1	1	1	1
gpt-4.1-2025-04-14	6	4	6	16	8	8	4	4
claude-sonnet-4-20250514	13	11	7	7	8	8	5	5
o4-mini-2025-04-16	13	12	10	3	17	17	20	20
claude-3-7-sonnet-20250219-thinking-32k	19	12	15	12	7	7	5	5
gpt-4o-2024-05-13	39	48	43	45	44	44	42	42

Table 10: LMArena performance rankings for Copilot Pro models (retrieved 2025-07-24 [83])

Model ID	Index Score		Reasoning Model
	Intelligence	Coding	
o4-mini	65.05	63.48	●
gemini-2-5-pro	64.63	61.46	●
gpt-4-1	46.77	41.90	○
claude-3-7-sonnet-thinking	46.98	43.79	●
claude-4-sonnet	45.72	41.07	○
gpt-4o	29.61	32.12	○

Note: gpt-4o = gpt-4o-2024-11-20 according to source [85]; filled circle: reasoning model; empty circle: non-reasoning model

Table 11: Artificial Analysis benchmark scores for Copilot Pro models (Intelligence and Coding Index benchmarks; retrieved 2025-08-05 [85])

4.2 AI Benchmarking

As new and improved LLMs are released at a rapid pace, benchmarking serves as an essential tool for evaluating model performance and identifying optimal models for specific applications. However, benchmark scores can be misleading and may not accurately reflect real-world performance due to several critical limitations [92][93]:

Benchmark contamination occurs when training data includes benchmark test cases, resulting in artificially inflated scores through memorization rather than actual understanding or capability.

Limited real-world applicability emerges when benchmarks focus on narrow, isolated tasks that fail to capture the complexity and variability of practical use cases.

Evaluation subjectivity affects open-ended tasks such as text summarization and complex reasoning, where automated evaluation methods may introduce systematic biases, while human evaluation often lacks consistency across assessors.

The benchmarking platforms LMArena and Artificial Analysis implement distinct methodologies to address these challenges in evaluating LLMs.

LMArena functions as an open platform that assesses language models based on crowdsourced human preferences [84]. Users are presented with responses from two anonymous models and select the one they find preferable in a pairwise comparison format. This dynamic setup reduces benchmark contamination by relying on live user-submitted questions rather than fixed datasets. It also mitigates evaluation subjectivity by collecting feedback from a broad and linguistically diverse user base.

In contrast, Artificial Analysis follows a more standardized approach, conducting evaluations across multiple established benchmark suites [94]. Its Intelligence Index aggregates model performance in reasoning, mathematics, and programming using weighted averages of existing benchmarks. The Coding Index specifically measures programming ability by combining results from several code generation benchmarks.

Both platforms emphasize transparency by openly documenting their evaluation procedures and using standardized testing conditions. They also implement robust validation methods to ensure the reliability of their assessments. LMArena applies statistical techniques to rank models based on user preferences. Artificial Analysis uses a combination of automated grading and LLM verification to validate responses. Together, these platforms offer complementary perspectives on model performance by balancing real-world user interaction with structured evaluation frameworks.

A comparison of benchmarking results from both platforms shows that model rankings are not identical across benchmarks. However, while the ranking order in LMArena does not exactly match that of Artificial Analysis, models that perform well on one benchmark generally also achieve higher rankings on the other (cf. Tab. 12, 13).

5 Chat Archiver Design and Implementation

Note: The following text and accompanying architectural diagram (Fig. 13) provide a high-level overview of my chat archiving utility. For brevity, I focus on the system’s design and core functionality rather than technical specifics. Complete source code and additional materials are available in dedicated repositories.

The main repository of the archiving utility, *chat-archiver*⁶⁰, contains the core system implementation. A separate repository for the chat conversion and Markdown “rendering” functionality⁶¹ is included as a git submodule [95].

The database container configuration is located in my *project* repository⁶², which contains various resources accumulated during this thesis with broader relevance across different project components. Since I also used the database for generic code optimization testing, I moved its container configuration to this shared repository during development.

Backups of the database are archived in the *project-db-dump* repository⁶³, and Markdown-converted chat logs are available in the *chats* repository⁶⁴.

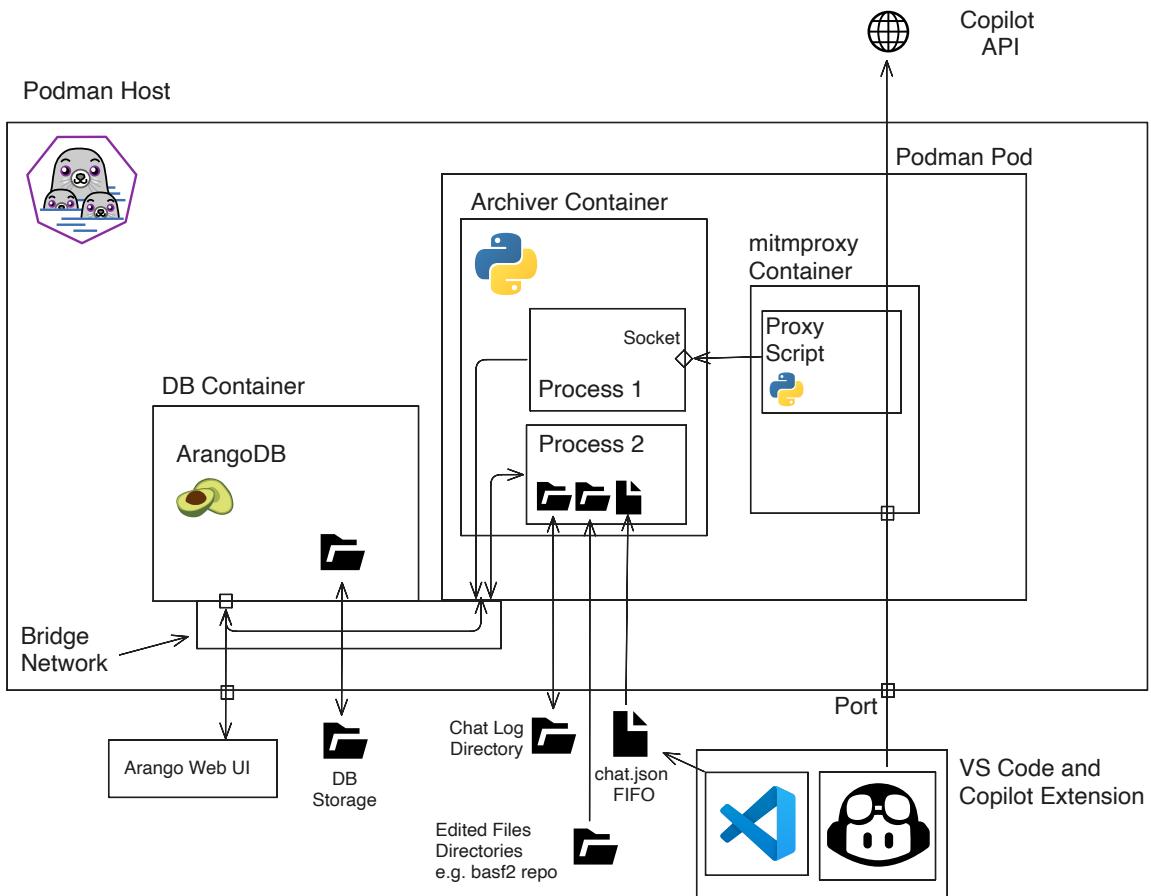


Figure 13: Architecture Diagram of the Chat Archiver⁶⁵

To support my work, I developed a chat archiver designed to back up and convert chat logs exported from the GitHub Copilot extension in Visual Studio Code. The goal was to transform raw JSON chat data into a human-readable, annotated Markdown format. Each chat entry is enriched with metadata, including the specific model used to generate the response, the runtime of each request, and file diffs for any edits performed by Copilot.

⁶⁰github.com/philipobi-bachelor/chat-archiver

⁶¹github.com/philipobi-bachelor/chat-renderer

⁶²github.com/philipobi-bachelor/project

⁶³github.com/philipobi-bachelor/project-db-dump

⁶⁴github.com/philipobi-bachelor/chats

⁶⁵Icon Sources: [96][97][98][99][100][101][102][103][104]

However, the exported JSON logs do not include model information by default. To recover this missing data, I implemented a more involved solution that intercepts the network traffic of the Copilot extension. This is achieved by routing all VS Code traffic through mitmproxy [105] (Man-In-The-Middle proxy).

This software allows to extract and process relevant data from the network stream with custom Python scripts that filter HTTP(S) requests passing through the proxy. I configured such a script to intercept only requests to Copilot API endpoints.

The entire system runs inside a containerized environment using Podman [106]. Within a Podman pod, mitmproxy operates in one container, while a second container runs the Python-based archiving script. In Podman, a pod is a facility that groups containers, allowing them for example to share the same network namespace.

The archiving script executes two core processes:

- 1. Listener Process:** This process receives relevant intercepted Copilot API data via a network socket, specifically the LLM identifier that generated a chat response along with the associated internal message ID. The mitmproxy Python user-script transmits this information after extracting it from HTTP requests to Copilot's chat completion API endpoint.
- 2. Archiver Process:** This second process reads data from a named pipe (FIFO) mounted into the container from the host system. When a JSON chat log is exported from VS Code and written to the FIFO, the process first saves the raw JSON document to the database. It then parses the chat data and converts it into a Markdown file.

The JSON chat logs include the internal Copilot message IDs, which allows the archiver to associate responses with their corresponding models by querying the database of previously stored model-ID pairs.

Additionally, the utility generates diffs for any file edits made during conversations in Copilot's Agent or Edit modes. For diff generation to work, edited files must be accessible from within the container. To ensure this, I mounted the necessary directories into the container in read-only mode, namely the basf2 directory during V0Fitter refactoring, and the directory containing the task files for generic code optimization testing.

The generated Markdown files are saved to a mounted output directory, allowing direct access from the host system.

For all database needs throughout the thesis, I used ArangoDB [107]. A key advantage of ArangoDB over traditional SQL databases is its document database functionality, which allows direct storage of JSON-like documents and flexible querying of their keys and values. This was particularly beneficial for storing and analyzing the chat log JSON documents and intercepted API responses.

I used ArangoDB not only for chat archiving purposes but also for storing benchmarking data related to the generic code optimization tests. The database therefore runs in a separate container outside the archiving pod, with persistent storage mounted from the host. To enable communication between the database container and the archiver pod, they are connected via a shared bridge network [108].

6 Summary

At the beginning of my research, I did not use AI tools frequently and was rather skeptical about their capabilities. Through the work on this thesis, I became significantly more familiar with LLMs and now have a better understanding of both their strengths and shortcomings.

Drawing from my experiences throughout the research process, I found that LLMs are overall very helpful for software optimization tasks.

For highly complex codebases such as basf2, the tests I performed showed mixed results. In my testing, LLMs appeared less suited for code generation tasks, at least with the models I evaluated. It is possible that more capable models could produce acceptable code for such complex systems. However, I found that LLMs proved very helpful for code analysis and understanding tasks. They excelled at codebase exploration, synthesizing and summarizing information from various files, explaining questions related to the code, and finding bugs.

For medium to low complexity coding tasks, my test results for generic code optimization indicated different performance patterns. I observed that LLMs are capable of generating functional and usable code, though the quality varied between models and specific examples. From my testing, I noticed that the more common the code pattern, the better the generated code generally was.

Beyond the formal testing scenarios, I benefited significantly from querying LLMs, mainly ChatGPT, during the development of test infrastructure and especially the V0Fitter refactoring process. LLMs provided valuable assistance with containerization-related topics for benchmarking the generic code optimization examples and for implementing my chat archiving tool. I also found that LLM responses were particularly useful for addressing general Python and Linux-related questions.

Overall, the main benefit of AI usage that I experienced was a significantly accelerated development speed: I could ask questions specific to my use case and get meaningful results, and my questions did not have to be perfectly formulated because LLMs can understand context and interpret vague or imprecise queries, which I found to be a significant advantage over conventional search engines. The responses to broad queries often pointed me in the right direction for more detailed research.

Among the models I compared, I had the best experience with Claude Sonnet 4, which consistently generated high-quality responses and, to some extent, code.

I also used AI extensively during the writing of my thesis, where I again had good experiences with Claude Sonnet 4. I accessed it primarily via Anthropic's chat website [109] and received overall better responses than from ChatGPT.

My use cases for AI during the thesis writing phase were mainly help with English formulations, searching through my LaTeX code to identify typographical and grammatical errors, detecting potential factual inconsistencies, help with implementing certain LaTeX functionality, and debugging LaTeX errors. Interestingly, I found that debugging LaTeX errors was almost as complex as debugging the code during the research phase of the thesis.

One factor that I perhaps should have paid more attention to during my tests is prompt quality and prompt engineering. Literature sources indicate that prompt composition has a strong impact on response quality [65], which I also observed during my tests. I believe the AI models might have performed better on some of my tests if the prompts had been more carefully constructed. However, I think that having to carefully compose prompts for a model to produce acceptable results reduces many of the benefits that LLMs offer, which are considerably rooted in the ability to use natural language queries to interact with the system.

While LLMs offer many advantages for software development and optimization tasks, they also have notable limitations and disadvantages. First, as observed from some responses throughout the course of this thesis, LLMs do make mistakes and can confidently provide incorrect answers. Also, having entire codebases generated by AI can lead to code that includes vulnerabilities, has inconsistent coding conventions, and is overall hard to maintain. Another concern I have about LLMs is their high environmental impact, mainly during training but also during inference.

Another issue that I consider significant is that AI is increasingly replacing the use of knowledge bases and

forums, such as Stack Overflow [110]. I find this problematic because Stack Overflow constitutes a transparent and human-readable knowledge base, which is likely where much of AI’s software development knowledge comes from. In contrast, AI’s knowledge is encoded in its parameters, and this information is not readily accessible due to both the closed-source nature of many popular LLMs and the fact that information is spread across billions of learned parameters with no way to extract it. As more people rely on LLMs for software development support, I believe that less knowledge will be documented in forums like Stack Overflow. Additionally, LLMs generally do not keep track of helpful responses, whereas on Stack Overflow, posts with many upvotes can generally be considered correct or helpful to many people. I think this might lead to a loss of publicly available knowledge in the long term.

7 Outlook

Given that AI technology will continue to evolve and demonstrates substantial potential across various domains, I believe it would be unwise not to engage with it, and explore potential applications for software optimization tasks.

While current AI models already prove helpful for software development, future models will likely improve further, making this a significant area of development for years to come.

A promising AI field for software optimization, in my opinion, is constraining AI models’ outputs, similar to the approach employed by Google’s AlphaDev system [111]. AlphaDev, based on the AlphaZero architecture that achieved world-class performance in chess, shogi, and Go [112], treated algorithm optimization as a strategic game. The system operated with predefined moves, where each move represented adding a specific assembly instruction to the program. Through this game-based approach, it was able to discover more efficient algorithms, such as a faster sorting algorithm for five-element sequences.

I find this approach to AI software optimization particularly elegant, as programming languages are inherently constrained by a predefined set of rules, making it possible to clearly determine whether an expression is valid or not.

This characteristic fundamentally differs from human language, which contains substantial nuance and ambiguity. LLMs excel at natural language tasks precisely because their extensive parameters and training on large text corpora enable them to capture subtle patterns and generalize beyond training examples. While this flexibility benefits natural language processing applications, it can be disadvantageous for rigid tasks such as programming, where precision and correctness are paramount.

References

- [1] The Belle II Collaboration. *Belle II Analysis Software Framework (basf2)*. Version: release-09-00-00. Source code available at <https://github.com/belle2/basf2>. Jan. 18, 2025. doi: [10.5281/zenodo.5574115](https://doi.org/10.5281/zenodo.5574115). URL: <https://software.belle2.org/>.
- [2] T. Kuhr et al. “The Belle II Core Software: Belle II Framework Software Group”. In: *Computing and Software for Big Science* 3.1 (Nov. 2018). ISSN: 2510-2044. doi: [10.1007/s41781-018-0017-9](https://doi.org/10.1007/s41781-018-0017-9). URL: <http://dx.doi.org/10.1007/s41781-018-0017-9>.
- [3] A. Danial et al. *cloc*. Version: 2.06. GitHub repository. June 25, 2025. URL: <https://github.com/AlDanial/cloc>.
- [4] E Kou et al. “The Belle II Physics Book”. In: *Progress of Theoretical and Experimental Physics* 2019.12 (Dec. 2019). ISSN: 2050-3911. doi: [10.1093/ptep/ptz106](https://doi.org/10.1093/ptep/ptz106). URL: <http://dx.doi.org/10.1093/ptep/ptz106>.
- [5] Werner Herr and B Muratori. “Concept of luminosity”. In: (2006). doi: [10.5170/CERN-2006-002.361](https://doi.org/10.5170/CERN-2006-002.361). URL: <https://cds.cern.ch/record/941318>.
- [6] T. Abe et al. “Belle II Technical Design Report”. In: (Nov. 2010). arXiv: [1011.0352 \[physics.ins-det\]](https://arxiv.org/abs/1011.0352).
- [7] Swagato Banerjee. “Searches for Lepton Flavor Violation in Tau Decays at Belle II”. In: *Universe* 8.9 (2022). ISSN: 2218-1997. doi: [10.3390/universe8090480](https://doi.org/10.3390/universe8090480). URL: <https://www.mdpi.com/2218-1997/8/9/480>.
- [8] Takanori Hara, Thomas Kuhr, and Yutaka Ushiroda. *Belle II Coordinate System and Guideline of Belle II Numbering Scheme*. Aug. 9, 2011. URL: <https://indico.mpp.mpg.de/event/2308/contributions/4092/attachments/3414/3799/Belle2NumberingScheme.pdf> (visited on 07/30/2025).
- [9] The Belle II Collaboration. *The Belle II Software Online Book*. Jan. 18, 2025. URL: <https://software.belle2.org/release-09-00-00/sphinx> (visited on 07/30/2025).
- [10] The Belle II Collaboration. *basf2 Doxygen Documentation*. Jan. 18, 2025. URL: <https://software.belle2.org/release-09-00-02/doxygen> (visited on 07/30/2025).
- [11] Rene Brun et al. *ROOT*. Version: 6-18-02. Aug. 2019. doi: [10.5281/zenodo.3895860](https://doi.org/10.5281/zenodo.3895860). URL: <https://doi.org/10.5281/zenodo.3895860>.
- [12] S. Agostinelli et al. “GEANT4 - A Simulation Toolkit”. In: *Nucl. Instrum. Meth. A* 506 (2003), pp. 250–303. doi: [10.1016/S0168-9002\(03\)01368-8](https://doi.org/10.1016/S0168-9002(03)01368-8).
- [13] The Belle II Collaboration. *basf2 Sphinx Documentation — Modules and Paths*. Jan. 18, 2025. URL: https://software.belle2.org/development/sphinx/framework/doc/modules_paths.html#id1 (visited on 07/30/2025).
- [14] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. 4th ed. Hoboken, NJ: Pearson, 2020. ISBN: 9780134610993.
- [15] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [16] Ashish Vaswani et al. *Attention Is All You Need*. 2023. arXiv: [1706.03762 \[cs.CL\]](https://arxiv.org/abs/1706.03762). URL: <https://arxiv.org/abs/1706.03762>.
- [17] Wayne Xin Zhao et al. “A Survey of Large Language Models”. In: *arXiv preprint arXiv:2303.18223* (2023). URL: [http://arxiv.org/abs/2303.18223](https://arxiv.org/abs/2303.18223).
- [18] Alec Radford and Karthik Narasimhan. “Improving Language Understanding by Generative Pre-Training”. In: 2018. URL: <https://api.semanticscholar.org/CorpusID:49313245>.
- [19] Tom B. Brown et al. *Language Models are Few-Shot Learners*. 2020. arXiv: [2005.14165 \[cs.CL\]](https://arxiv.org/abs/2005.14165). URL: <https://arxiv.org/abs/2005.14165>.
- [20] Jie Huang and Kevin Chen-Chuan Chang. *Towards Reasoning in Large Language Models: A Survey*. 2023. arXiv: [2212.10403 \[cs.CL\]](https://arxiv.org/abs/2212.10403). URL: <https://arxiv.org/abs/2212.10403>.
- [21] DeepSeek-AI et al. *DeepSeek-R1: Incentivizing Reasoning Capability in LLMs via Reinforcement Learning*. 2025. arXiv: [2501.12948 \[cs.CL\]](https://arxiv.org/abs/2501.12948). URL: <https://arxiv.org/abs/2501.12948>.
- [22] Patrick Lewis, Ethan Perez, et al. *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*. 2021. arXiv: [2005.11401 \[cs.CL\]](https://arxiv.org/abs/2005.11401). URL: <https://arxiv.org/abs/2005.11401>.
- [23] Tula Masterman et al. *The Landscape of Emerging AI Agent Architectures for Reasoning, Planning, and Tool Calling: A Survey*. 2024. arXiv: [2404.11584 \[cs.AI\]](https://arxiv.org/abs/2404.11584). URL: <https://arxiv.org/abs/2404.11584>.
- [24] GitHub. *GitHub Copilot Product Page*. 2025. URL: <https://github.com/features/copilot> (visited on 07/30/2025).
- [25] GitHub. *GitHub Copilot Documentation*. 2025. URL: <https://docs.github.com/en/copilot> (visited on 07/30/2025).
- [26] Mark Chen et al. *Evaluating Large Language Models Trained on Code*. 2021. arXiv: [2107.03374 \[cs.LG\]](https://arxiv.org/abs/2107.03374). URL: <https://arxiv.org/abs/2107.03374>.
- [27] Maxwell Zeff. *GitHub Copilot crosses 20M all-time users*. July 30, 2025. URL: <https://techcrunch.com/2025/07/30/github-copilot-crosses-20-million-all-time-users/> (visited on 07/30/2025).
- [28] Microsoft. *Visual Studio Code*. Version: 1.102.2. July 22, 2025. URL: <https://code.visualstudio.com/>.
- [29] Microsoft. *Microsoft Remote Development Pack for VS Code*. Version: 0.26.0. Visual Studio Code extension pack. May 3, 2025. URL: <https://github.com/microsoft/vscode-remote-release>.
- [30] GitHub. *GitHub Copilot Extension for VS Code*. Version: 1.350.0. Visual Studio Code extension providing inline coding suggestions. July 29, 2025. URL: <https://marketplace.visualstudio.com/items?itemName=GitHub.copilot>.

- [31] GitHub and Microsoft. *GitHub Copilot Chat Extension for VS Code*. Version: 0.29.1. Visual Studio Code extension providing AI chat functionality. July 16, 2025. URL: <https://github.com/microsoft/vscode-copilot-chat>.
- [32] Tom Warren. *GitHub Copilot will support models from Anthropic, Google, and OpenAI*. Oct. 29, 2024. URL: <https://www.theverge.com/2024/10/29/24282544/github-copilot-multi-model-anthropic-google-openai-github-spark-announcement> (visited on 07/30/2025).
- [33] Model Context Protocol Community and Anthropic. *Model Context Protocol Specification*. Version: 2025-06-18. Specification source code available at <https://github.com/modelcontextprotocol/modelcontextprotocol>. June 18, 2025. URL: <https://modelcontextprotocol.io/overview>.
- [34] GitHub. *GitHub Student Developer Pack*. 2025. URL: <https://education.github.com/pack> (visited on 07/28/2025).
- [35] The Belle II Collaboration. *basf2 Doxygen Documentation — V0Fitter*. Jan. 18, 2025. URL: https://software.belle2.org/release-09-00-00/doxygen/classBelle2_1_1V0Fitter.html (visited on 07/15/2025).
- [36] The Belle II Collaboration. *basf2 Sphinx Documentation — V0 Finding*. Jan. 18, 2025. URL: <https://software.belle2.org/release-09-00-00/sphinx/tracking/doc/specials.html#v0-finding> (visited on 07/15/2025).
- [37] The Belle II Collaboration. *basf2 Doxygen Documentation — V0FinderModule*. Jan. 18, 2025. URL: https://software.belle2.org/release-09-00-00/doxygen/classBelle2_1_1V0FinderModule.html (visited on 07/15/2025).
- [38] Johannes Rauch and Tobias Schlüter. “GENFIT — a Generic Track-Fitting Toolkit”. In: *Journal of Physics: Conference Series* 608.1 (Apr. 2015). DOI: [10.1088/1742-6596/608/1/012042](https://dx.doi.org/10.1088/1742-6596/608/1/012042). URL: <https://dx.doi.org/10.1088/1742-6596/608/1/012042>.
- [39] Wolfgang Waltenberger, Winfried Mitaroff, and Fabian Moser. “RAVE—a Detector-independent vertex reconstruction toolkit”. In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 581.1 (2007). VCI 2007, pp. 549–552. ISSN: 0168-9002. DOI: [10.1016/j.nima.2007.08.048](https://doi.org/10.1016/j.nima.2007.08.048). URL: <https://www.sciencedirect.com/science/article/pii/S0168900207017196>.
- [40] The Belle II Collaboration. *basf2 Doxygen Documentation — Utils::Timer*. Jan. 18, 2025. URL: https://software.belle2.org/release-09-00-00/doxygen/classBelle2_1_1Utils_1_1Timer.html (visited on 07/15/2025).
- [41] Jim Pivarski et al. *Awkward Array*. Oct. 2018. DOI: [10.5281/zenodo.4341376](https://doi.org/10.5281/zenodo.4341376).
- [42] Charles R. Harris et al. “Array programming with NumPy”. In: *Nature* 585 (2020), pp. 357–362. DOI: [10.1038/s41586-020-2649-2](https://doi.org/10.1038/s41586-020-2649-2).
- [43] The pandas development team. *Pandas*. Version: v2.2.2. Apr. 10, 2025. DOI: [10.5281/zenodo.3509134](https://doi.org/10.5281/zenodo.3509134). URL: <https://doi.org/10.5281/zenodo.3509134>.
- [44] Josef Weidendorfer, Markus Kowarschik, and Carsten Trinitis. “A Tool Suite for Simulation Based Analysis of Memory Access Behavior”. In: *Computational Science - ICCS 2004*. Ed. by Marian Bubak et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 440–447. ISBN: 978-3-540-24688-6.
- [45] Nicholas Nethercote and Julian Seward. “Valgrind: a framework for heavyweight dynamic binary instrumentation”. In: *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*. PLDI ’07. San Diego, California, USA: Association for Computing Machinery, 2007, pp. 89–100. ISBN: 9781595936332. DOI: [10.1145/1250734.1250746](https://doi.org/10.1145/1250734.1250746). URL: <https://doi.org/10.1145/1250734.1250746>.
- [46] The Valgrind Developers. *Valgrind User Manual*. May 20, 2025. URL: <https://valgrind.org/docs/manual/manual.html> (visited on 07/20/2025).
- [47] John L Hennessy and David A Patterson. *Computer Architecture*. en. 6th ed. The Morgan Kaufmann Series in Computer Architecture and Design. Oxford, England: Morgan Kaufmann, Nov. 2017.
- [48] José Fonseca et al. *gprof2dot*. Version: 2025.04.14. GitHub repository. Apr. 14, 2025. URL: <https://github.com/jrfonseca/gprof2dot>.
- [49] John Ellson et al. “Graphviz and Dynagraph – Static and Dynamic Graph Drawing Tools”. In: (Jan. 2004), pp. 127–148. DOI: [10.1007/978-3-642-18638-7_6](https://doi.org/10.1007/978-3-642-18638-7_6).
- [50] The Belle II Collaboration. *basf2 Doxygen Documentation — RecoTrack*. Jan. 18, 2025. URL: https://software.belle2.org/release-09-00-00/doxygen/classBelle2_1_1RecoTrack.html (visited on 07/15/2025).
- [51] The Belle II Collaboration. *basf2 Doxygen Documentation — Track*. Jan. 18, 2025. URL: https://software.belle2.org/release-09-00-00/doxygen/classBelle2_1_1Track.html (visited on 07/15/2025).
- [52] The Belle II Collaboration. *basf2 Doxygen Documentation — RecoTrackGenfitAccess*. Jan. 18, 2025. URL: https://software.belle2.org/release-09-00-00/doxygen/classBelle2_1_1RecoTrackGenfitAccess.html (visited on 07/15/2025).
- [53] The Belle II Collaboration. *basf2 Doxygen Documentation — V0*. Jan. 18, 2025. URL: https://software.belle2.org/release-09-00-00/doxygen/classBelle2_1_1V0.html (visited on 07/15/2025).

- [54] The Belle II Collaboration. *basf2 Doxygen Documentation — TrackFitter::fit*. Jan. 18, 2025. URL: https://software.belle2.org/release-09-00-00/doxygen/classBelle2_1_1TrackFitter.html#abb4a849613f8a34fc298b4c41706c15b (visited on 07/30/2025).
- [55] The Belle II Collaboration. *v0ValidationGenerateSample.py*. Version: release-09-00-00. basf2 V0 Validation Script. Jan. 18, 2025. URL: <https://github.com/belle2/basf2/blob/release-09-00-00/tracking/validation/v0ValidationGenerateSample.py>.
- [56] The Belle II Collaboration. *KShortGenSimNoBkg.py*. Version: release-09-00-00. basf2 KShort MC Generation Script. Jan. 18, 2025. URL: <https://github.com/belle2/basf2/blob/release-09-00-00/validation/validation/KShortGenSimNoBkg.py>.
- [57] The Belle II Collaboration. *basf2 Sphinx Documentation — ParticleGun Module*. Jan. 18, 2025. URL: <https://software.belle2.org/release-09-00-00/sphinx/generators/doc/modules.html#module-ParticleGun> (visited on 07/30/2025).
- [58] The Belle II Collaboration. *v0ValidationCreatePlots.py*. Version: release-09-00-00. basf2 V0 Validation Plot Generation Script. Jan. 18, 2025. URL: <https://github.com/belle2/basf2/blob/release-09-00-00/tracking/validation/v0ValidationCreatePlots.py>.
- [59] The Belle II Collaboration. *b2validation-plots*. Version: release-09-00-00. basf2 Validation Plot Comparison Script. Jan. 18, 2025. URL: <https://github.com/belle2/basf2/blob/release-09-00-00/validation/tools/b2validation-plots>.
- [60] Anthropic. *Claude 4*. Version: Sonnet 4. LLM developed by Anthropic. May 22, 2025. URL: <https://www.anthropic.com/news/clause-4>.
- [61] Google Gemini Team. *Gemini 2.5*. Version: 2.5 Pro. LLM developed by Google. Mar. 25, 2025. URL: <https://blog.google/technology/google-deepmind/gemini-model-thinking-updates-march-2025/#gemini-2-5-thinking>.
- [62] openAI. *OpenAI o4-mini*. Version: o4-mini. LLM developed by openAI. Apr. 16, 2025. URL: <https://openai.com/index/introducing-o3-and-o4-mini/>.
- [63] openAI. *GPT-4o*. Version: GPT-4o. LLM developed by openAI. May 13, 2025. URL: <https://openai.com/index/hello-gpt-4o/>.
- [64] Teodora Musatoiu. *Enhance Your Prompts with Meta-Prompting*. Oct. 23, 2024. URL: https://cookbook.openai.com/examples/enhance_your_prompts_with_meta_prompting (visited on 07/29/2025).
- [65] Lucas Jasper Jacobsen and Kira Elena Weber. “The Promises and Pitfalls of Large Language Models as Feedback Providers: A Study of Prompt Engineering and the Quality of AI-Driven Feedback”. In: *AI* 6.2 (2025). ISSN: 2673-2688. DOI: [10.3390/ai6020035](https://doi.org/10.3390/ai6020035). URL: <https://www.mdpi.com/2673-2688/6/2/35>.
- [66] The Doxygen Developers. *Doxygen*. Version: 1.14.0. A documentation generator. May 24, 2025. URL: <https://github.com/doxygen/doxygen>.
- [67] OpenAI. *ChatGPT*. Chat interface accessing various OpenAI LLMs, developed by OpenAI. 2025. URL: <https://chatgpt.com/>.
- [68] The Belle II Collaboration. *coding_conventions.rst*. Version: release-09-00-00. basf2 Coding Conventions. Jan. 18, 2025. URL: https://github.com/belle2/basf2/blob/release-09-00-00/framework/doc/development/coding_conventions.rst.
- [69] Anthropic. *Claude Sonnet 3.7*. Version: Sonnet 3.7 Thinking. LLM developed by Anthropic. Feb. 24, 2025. URL: <https://www.anthropic.com/news/clause-3-7-sonnet>.
- [70] Linux Kernel Community. *OverlayFS — Overlay Filesystem*. A union mount filesystem implementation in the Linux kernel. 2024. URL: <https://www.kernel.org/doc/html/latest/filesystems/overlayfs.html> (visited on 07/29/2025).
- [71] Giuseppe Scrivano et al. *fuse-overlayfs*. Version: 1.13. FUSE implementation for OverlayFS. 2023. URL: <https://github.com/containers/fuse-overlayfs>.
- [72] Marcin Bajer. “Securing and Hardening Embedded Linux Devices - case study based on NXP i.MX6 Platform”. In: *2022 9th International Conference on Future Internet of Things and Cloud (FiCloud)*. 2022, pp. 181–189. DOI: [10.1109/FiCloud57274.2022.00032](https://doi.org/10.1109/FiCloud57274.2022.00032).
- [73] Microsoft. *Debug Adapter Protocol*. Specification for communicating between development tools and debuggers. 2024. URL: <https://microsoft.github.io/debug-adapter-protocol/> (visited on 07/29/2025).
- [74] Microsoft. *Microsoft C/C++ extension for VS Code*. Version: 1.26.3. Visual Studio Code extension. June 25, 2025. URL: <https://github.com/microsoft/vscode-cpptools>.
- [75] Free Software Foundation. *GNU Debugger (GDB)*. Version: 14.2. Debugger for programs written in C, C++, and other languages. 2024. URL: <https://www.gnu.org/software/gdb/>.
- [76] Bram Moolenaar et al. *Vim: The Editor*. Version: 8.2. Highly configurable text editor. 2019. URL: <https://www.vim.org/>.
- [77] LLVM Project. *Visual Studio Code extension for clangd*. Version: 0.2.0. Visual Studio Code extension. June 10, 2025. URL: <https://github.com/clangd/vscode-clangd>.
- [78] LLVM Project. *clangd*. Version: 18.1. C/C++ language server based on Clang. 2024. URL: <https://clangd.llvm.org/>.
- [79] Pionxzh et al. *ChatGPT Exporter*. Version: 2.29.1. GitHub repository. July 25, 2025. URL: <https://github.com/pionxzh/chatgpt-exporter>.
- [80] The Open Group. *Regular Expressions*. 1997. URL: <https://pubs.opengroup.org/onlinepubs/7908799/xbd/re.html> (visited on 08/01/2025).

- [81] Firas Dib et al. *Regex101*. 2025. URL: <https://regex101.com/>.
- [82] Martin Leitner-Ankerl et al. *Nanobench*. Version: 4.3.11. GitHub repository. Feb. 16, 2023. URL: <https://github.com/martinus/nanobench>.
- [83] Wei-Lin Chiang et al. *LMArena*. 2025. URL: <https://lmarena.ai/leaderboard> (visited on 07/24/2025).
- [84] Wei-Lin Chiang et al. *Chatbot Arena: An Open Platform for Evaluating LLMs by Human Preference*. 2024. arXiv: [2403.04132 \[cs.AI\]](#). URL: <https://arxiv.org/abs/2403.04132>.
- [85] Artificial Analysis. *Artificial Analysis AI Models Comparison*. 2025. URL: <https://artificialanalysis.ai/models> (visited on 08/03/2025).
- [86] openAI. *GPT-4.1*. Version: GPT-4.1. LLM developed by openAI. Apr. 14, 2025. URL: <https://openai.com/index/gpt-4-1/>.
- [87] openAI. *OpenAI o3*. Version: o4-mini. LLM developed by openAI. Apr. 16, 2025. URL: <https://openai.com/index/introducing-o3-and-o4-mini/>.
- [88] Anthropic. *Claude 4*. Version: Opus 4. LLM developed by Anthropic. May 22, 2025. URL: <https://www.anthropic.com/news/clause-4>.
- [89] Anthropic. *Claude Sonnet 3.5*. Version: Sonnet 3.5. LLM developed by Anthropic. June 21, 2024. URL: <https://www.anthropic.com/news/clause-3-5-sonnet>.
- [90] Google Gemini Team. *Gemini 2.0*. Version: 2.0 Flash. LLM developed by Google. Dec. 11, 2024. URL: <https://blog.google/technology/google-deepmind/google-gemini-ai-update-december-2024/>.
- [91] GitHub. *Supported AI models in Copilot*. 2025. URL: <https://docs.github.com/en/copilot/reference/ai-models/supported-models> (visited on 08/05/2025).
- [92] Chunyuan Deng et al. *Unveiling the Spectrum of Data Contamination in Language Models: A Survey from Detection to Remediation*. 2024. arXiv: [2406.14644 \[cs.CL\]](#). URL: <https://arxiv.org/abs/2406.14644>.
- [93] Maria Eriksson et al. *Can We Trust AI Benchmarks? An Interdisciplinary Review of Current Issues in AI Evaluation*. 2025. arXiv: [2502.06559 \[cs.AI\]](#). URL: <https://arxiv.org/abs/2502.06559>.
- [94] Artificial Analysis. *Artificial Analysis Benchmarking Methodology*. 2025. URL: <https://artificialanalysis.ai/methodology> (visited on 08/03/2025).
- [95] Scott Chacon and Ben Straub. *Git Tools - Submodules*. Chapter 7.11. URL: <https://git-scm.com/book/en/v2/Git-Tools-Submodules> (visited on 08/05/2025).
- [96] *ArangoDB Icon*. Icon resource. Streamline HQ. URL: <https://www.streamlinehq.com/icons/download/arangodb-icon--31103> (visited on 08/07/2025).
- [97] *GitHub Copilot Icon*. Icon resource. UXWing. URL: <https://uxwing.com/github-copilot-icon/> (visited on 08/07/2025).
- [98] *Folder Open SVG Icon*. SVG icon resource. SVG Repo. URL: <https://www.svgrepo.com/svg/535397/folder-open> (visited on 08/07/2025).
- [99] *File SVG Icon*. SVG icon resource. SVG Repo. URL: <https://www.svgrepo.com/svg/535388/file> (visited on 08/07/2025).
- [100] *Python Logo (No Text)*. SVG logo file. Wikimedia Commons. URL: <https://commons.wikimedia.org/wiki/File:Python-logo-notext.svg> (visited on 08/07/2025).
- [101] *Podman Icon*. Icon resource. Tech Icons. URL: <https://techicons.dev/icons/podman> (visited on 08/07/2025).
- [102] *Internet SVG Icon*. SVG icon resource. SVG Repo. URL: <https://www.svgrepo.com/svg/478288/internet> (visited on 08/07/2025).
- [103] *Mitmproxy Logo*. SVG logo file. Wikimedia Commons. URL: https://commons.wikimedia.org/wiki/File:Mitmproxy_logo.svg (visited on 08/07/2025).
- [104] *VS Code Logo*. SVG logo file. Wikimedia Commons. URL: https://commons.wikimedia.org/wiki/File:Visual_Studio_Code_1.35_icon.svg (visited on 08/07/2025).
- [105] The mitmproxy Developer Team. *mitmproxy*. Version: 12.1.1. GitHub repository. May 25, 2025. URL: <https://github.com/mitmproxy/mitmproxy>.
- [106] Red Hat, Inc. et al. *Podman*. Version: 5.0. Daemonless container engine for developing, managing, and running OCI containers. 2025. URL: <https://podman.io/>.
- [107] The ArangoDB Developer Team. *ArangoDB*. Version: 3.12.5.2. GitHub repository. July 30, 2025. URL: <https://github.com/arangodb/arangodb>.
- [108] *Bridge network driver*. Podman implementation similar but missing documentation. Docker Inc. URL: <https://docs.docker.com/engine/network/drivers/bridge/> (visited on 08/05/2025).
- [109] Anthropic. *Claude Chat*. 2025. URL: <https://claude.ai>.
- [110] Drew Breunig. *StackOverflow's LLM-Accelerated Decline*. May 16, 2025. URL: <https://www.dbreunig.com/2025/05/16/stackoverflow-s-decline.html> (visited on 08/06/2025).
- [111] Daniel Mankowitz et al. “Faster sorting algorithms discovered using deep reinforcement learning”. In: *Nature* 618 (June 2023), pp. 257–263. DOI: [10.1038/s41586-023-06004-9](#).
- [112] David Silver et al. *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*. 2017. arXiv: [1712.01815 \[cs.AI\]](#). URL: <https://arxiv.org/abs/1712.01815>.

8 Appendix

Model	Overall	Hard			Instruction Following	Longer Query
		Prompts	Coding	Math		
gemini-2.5-pro	1	1	1	1	1	1
o3-2025-04-16	2	2	1	1	2	6
chatgpt-4o-latest-20250326	3	2	1	7	2	1
claude-opus-4-20250514-thinking-16k	6	3	1	5	2	1
grok-4-0709	3	2	2	1	2	2
gpt-4.5-preview-2025-02-27	3	4	2	6	2	2
MoonshotAI kimi-k2-0711-preview	5	2	2	7	7	5
claude-opus-4-20250514	6	4	2	5	2	1
claude-sonnet-4-20250514-thinking-32k	11	4	2	5	4	2
deepseek-r1-0528	6	4	4	6	8	9
qwen3-235b-a22b-no-thinking	13	10	6	6	14	8
gpt-4.1-2025-04-14	6	4	6	16	8	4
claude-sonnet-4-20250514	13	11	7	7	8	5
deepseek-v3-0324	13	14	8	17	14	13
grok-3-preview-02-24	7	7	9	13	8	5
o4-mini-2025-04-16	13	12	10	3	17	20
deepseek-r1	13	11	10	6	8	13
gpt-4.1-mini-2025-04-14	22	21	12	25	17	15
qwen3-32b	34	28	12	4	26	21
mistral-medium-2505	19	15	12	27	18	14
qwen3-235b-a22b	24	21	12	5	23	17
claude-3-7-sonnet-20250219-thinking-32k	19	12	15	12	7	5
o3-mini-high	27	15	15	7	20	21
gemini-2.5-flash	6	6	15	3	5	5
o1-2024-12-17	13	12	15	6	8	9

Table 12: Top 25 models by LMArena coding performance ranking (retrieved 2025-07-24 [83])

Model ID	Index Score		Reasoning Model
	Intelligence	Coding	
grok-4	67.52	63.81	●
o4-mini	65.05	63.48	●
gemini-2-5-pro	64.63	61.46	●
o3	67.07	59.69	●
deepseek-r1-0528	58.74	58.66	●
o3-mini-high	55.46	56.61	●
claude-4-sonnet-thinking	58.83	52.77	●
claude-4-opus-thinking	55.34	51.71	●
qwen3-235b-a22b-instruct-reasoning	48.42	51.08	●
deepseek-r1-0120	49.93	48.70	●
claude-4-opus	46.58	47.52	○
gpt-4-1-mini	42.25	44.32	○
qwen3-235b-a22b-instruct-2507	50.74	44.19	○
claude-3-7-sonnet-thinking	46.98	43.79	●
gpt-4-1	46.77	41.90	○
claude-4-sonnet	45.72	41.07	○
grok-3	39.92	39.67	○
gpt-4o-chatgpt-03-25	39.52	39.56	○
gemini-2-5-flash	47.26	39.29	○
deepseek-v3-0324	43.99	38.15	○
mistral-medium-3	38.59	36.55	○
kimi-k2	48.56	36.52	○
qwen3-235b-a22b-instruct	33.33	32.07	○
qwen3-32b-instruct	29.82	28.40	○
gpt-4-5	42.41	-	○

Note: filled circle: reasoning model; empty circle: non-reasoning model

Table 13: Artificial Analysis Intelligence and Coding Index scores for LMArena's top 25 coding models (cf. Tab. 12; retrieved 2025-08-05 [85])

Task Number / Description	bl	s/bl	s?	c-s-4	g-2.5-p	gpt-4o	o4-mini
1 Ineff. string concat. in loop	203 μ s	1.9	○	● 0.3	● -0.1	● -0.1	○ 0.0
2 Reserving vector capacity	57 μ s	0.7	○	● -0.7	○ 0.0	○ 0.0	○ 0.1
3 Excessive stream flushing	518 μ s	1.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
4 Passing/returning vector by value	11 μ s	0.0	○	● -0.5	● -0.5	○ 0.0	● -0.5
5 Repeated lookups with std::find	618 μ s	0.2	●	● 0.0	● 0.0	● 0.0	● -1.0
6 Passing shared_ptr by value	231 μ s	1.3	○	● 0.0	○ 0.0	○ 0.0	○ 0.0
7 Temporary objects and emplace_back	30 μ s	0.0	●	● 0.0	● 0.0	○ 0.0	● 0.0
8 Duplicate map lookups	425 μ s	0.3	○	● -2.7	● -1.5	○ 0.0	● -2.7
9 Bubble sort vs. std::sort	60 μ s	3.6	●	● 0.0	● 0.0	○ 3.6	● 0.0
10 Reading file one char at a time	142 μ s	0.7	○	○ 0.0	● 0.0	○ 0.0	○ 0.0
11 pow for integer powers of 2	679 ns	0.2	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
12 Repeated calls to vector::erase	15 μ s	1.8	○	● -0.1	● -0.1	○ 0.0	○ 0.0
13 Ineff. str to int conversion	912 ns	0.2	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
14 std::list for random access	25 μ s	2.6	○	● -0.1	○ 0.0	○ 0.0	○ 0.0
15 Passing large objects by value	163 μ s	0.4	○	○ 0.0	○ 0.0	● 0.0	○ 0.0
16 High contention on atomic	170 μ s	2.9	○	○ -0.1	○ -0.1	○ -0.1	○ -0.1
17 Using std::map when order not needed	8 ms	0.4	●	● 0.0	● 0.0	○ 0.4	○ 0.4
18 Using temp. string for comparison	470 μ s	0.8	○	● -2.8	○ 0.0	○ 0.0	○ 0.0
19 Inefficient map iteration	1 ms	0.9	○	● -3.1	○ 0.0	○ 0.0	● -3.1
20 Iteration with auto vs. auto&	207 μ s	0.9	○	● -2.4	○ 0.0	○ 0.0	○ 0.0
21 Opening/closing file in loop	74 μ s	2.1	●	● 0.0	● 0.0	○ 2.0	● 0.0
22 std::accumulate with ineff. operation	37 μ s	0.7	○	○ 0.0	● 0.0	○ 0.0	○ 0.0
23 Sorting large list in place	9 ms	0.4	○	○ 0.0	● 0.4	○ 0.0	○ 0.0
24 Loop vs. std::accumulate	334 μ s	0.0	●	● 0.0	● 0.0	○ 0.0	○ 0.0
25 Ineff. string concat. in loop	338 μ s	2.6	○	○ 0.0	● -0.4	○ 0.0	○ 0.0
26 Virtual function in loop	2 ms	0.0	●	● 0.0	● 0.0	○ 0.0	○ 0.0
27 shared_ptr for owned member	8 μ s	0.2	●	○ 0.2	○ 0.2	○ 0.2	● 0.2
28 Temporary objects and emplace_back	276 μ s	0.3	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
29 Re-calculating value in loop	1 ms	0.0	●	● -3.0	○ 0.0	○ 0.0	● -3.0
30 std::vector for fixed-size array	711 ns	3.3	●	● 0.0	● 0.0	○ 3.3	● 0.0
31 Manual loop to fill container	285 μ s	0.0	○	● -0.3	○ 0.0	○ 0.0	○ 0.0
32 Repeated object creation in loop	44 μ s	1.5	●	● -1.8	○ 1.5	○ 1.5	○ 1.5
33 Repeated stream setup in loop	20 ms	0.1	○	● -4.3	○ 0.0	○ 0.0	○ 0.0
34 std::set for simple existence checks	3 ms	0.7	●	● -3.5	● 0.0	○ 0.7	● -3.5
35 Ineff. string splitting	599 μ s	0.6	●	● -2.7	● 0.4	○ 0.6	○ 0.6
36 Manual vs. range container init.	492 μ s	0.7	●	● -1.5	○ 0.7	○ 0.7	○ 0.7
37 Returning large object by value	104 μ s	0.3	○	○ 0.0	● 0.4	○ 0.0	○ 0.0
38 std::map for int keys	926 μ s	0.2	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
39 Redundant loop bounds check	3 μ s	0.3	●	● 0.0	● 0.0	○ 0.3	○ 0.3
40 Div. and mod. for powers of 2	351 μ s	0.0	●	● 0.0	○ 0.0	○ 0.0	○ 0.0
41 std::map for enum keys	694 ns	0.1	●	● 0.1	○ 0.1	○ 0.1	○ 0.1
42 Ineff. string filtering	11 μ s	1.7	●	● 0.0	● 0.0	○ 1.7	● -0.5
43 map vs. vector for dense int keys	4 μ s	2.5	●	● 2.2	● 2.2	○ 2.4	○ 2.4
44 Column-major matrix traversal	227 ms	0.7	●	● 0.0	● 0.0	● 0.0	● 0.0
45 std::function overhead	23 ms	0.2	○	● 0.0	● 0.0	○ 0.0	○ 0.0
46 Modifying front of vector in loop	8 μ s	2.4	●	● 0.3	● 0.0	○ 2.4	● -0.2
47 Linear search on sorted data	3 μ s	0.4	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
48 Recompiling regex repeatedly	13 μ s	0.5	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
49 std::any when type is known	2 μ s	0.8	●	● -0.1	○ 0.8	○ 0.9	○ 0.8
50 Fibonacci without memoization	825 ns	5.3	○	● 0.0	● 0.0	○ 0.0	○ 0.0

Table 14: Copilot optimization results for Test File 1 (benchmarks compiled with -O3)

Task Number / Description	bl	s/bl	s?	c-s-4	g-2.5-p	gpt-4o	o4-mini
1 Ineff. string concat. in loop	203 μ s	1.9	●	● 0.3	● 0.3	● 0.3	● -0.1
2 Reserving vector capacity	57 μ s	0.7	●	● 0.0	● 0.0	● 0.0	● 0.0
3 Excessive stream flushing	518 μ s	1.0	●	○ 1.0	● 0.0	○ 1.0	○ 1.0
4 Passing/returning vector by value	11 μ s	0.0	●	○ -0.4	● -0.4	○ 0.0	○ 0.0
5 Repeated lookups with std::find	618 μ s	0.2	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
6 Passing shared_ptr by value	231 μ s	1.3	●	● 0.0	○ 1.3	○ 1.3	○ 1.3
7 Temporary objects and emplace_back	30 μ s	0.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
8 Duplicate map lookups	425 μ s	0.3	●	● -2.7	● 0.0	○ 0.3	○ 0.3
9 Bubble sort vs. std::sort	60 μ s	3.6	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
10 Reading file one char at a time	142 μ s	0.7	●	● 0.0	● 0.0	○ 0.8	● 0.0
11 pow for integer powers of 2	679 ns	0.2	●	● 0.0	● 0.0	○ 0.2	○ 0.2
12 Repeated calls to vector::erase	15 μ s	1.8	●	● -0.1	● -0.1	● -0.1	● 0.0
13 Ineff. str to int conversion	912 ns	0.2	●	● 0.0	● 0.0	○ 0.3	○ 0.2
14 std::list for random access	25 μ s	2.6	●	● 0.0	● 0.0	○ 2.7	○ 2.6
15 Passing large objects by value	163 μ s	0.4	●	● 0.0	● 0.0	● 0.1	● 0.0
16 High contention on atomic	170 μ s	2.9	●	○ 2.9	○ 3.0	● 3.0	○ 2.9
17 Using std::map when order not needed	8 ms	0.4	○	○ -0.1	○ -0.1	○ 0.0	○ 0.0
18 Using temp. string for comparison	470 μ s	0.8	●	● 0.0	● 0.0	○ 0.8	● 0.0
19 Inefficient map iteration	1 ms	0.9	●	● -3.0	● -3.1	○ 0.9	○ 0.9
20 Iteration with auto vs. auto&	207 μ s	0.9	●	● 0.0	● -2.4	○ 0.9	● 0.0
21 Opening/closing file in loop	74 μ s	2.1	○	○ 0.0	○ -0.1	○ 0.0	○ 0.0
22 std::accumulate with ineff. operation	37 μ s	0.7	●	○ 0.7	● 0.0	○ 0.7	○ 0.7
23 Sorting large list in place	9 ms	0.4	●	○ 0.4	● -0.2	○ 0.4	○ 0.4
24 Loop vs. std::accumulate	334 μ s	0.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
25 Ineff. string concat. in loop	338 μ s	2.6	●	● -2.1	● -2.1	● 0.0	● -2.1
26 Virtual function in loop	2 ms	0.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
27 shared_ptr for owned member	8 μ s	0.2	○	○ 0.0	○ -0.1	○ 0.0	○ -0.1
28 Temporary objects and emplace_back	276 μ s	0.3	●	● 0.0	● 0.0	○ 0.3	○ 0.1
29 Re-calculating value in loop	1 ms	0.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
30 std::vector for fixed-size array	711 ns	3.3	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
31 Manual loop to fill container	285 μ s	0.0	●	● -0.3	○ 0.0	○ 0.0	● -0.3
32 Repeated object creation in loop	44 μ s	1.5	○	○ -1.8	○ 0.1	○ 0.0	○ 0.0
33 Repeated stream setup in loop	20 ms	0.1	●	● -4.3	● -4.3	○ 0.1	● -4.3
34 std::set for simple existence checks	3 ms	0.7	○	○ -3.4	○ 0.0	○ 0.0	○ 0.0
35 Ineff. string splitting	599 μ s	0.6	○	○ -2.8	○ 0.0	○ 0.0	○ 0.0
36 Manual vs. range container init.	492 μ s	0.7	○	○ -1.4	○ 0.0	○ 0.0	○ 0.0
37 Returning large object by value	104 μ s	0.3	●	○ 0.3	○ 0.4	○ 0.3	● -0.2
38 std::map for int keys	926 μ s	0.2	●	○ 0.2	○ 0.2	○ 0.2	○ 0.2
39 Redundant loop bounds check	3 μ s	0.3	○	○ -0.2	○ 0.1	○ 0.0	○ 0.0
40 Div. and mod. for powers of 2	351 μ s	0.0	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
41 std::map for enum keys	694 ns	0.1	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
42 Ineff. string filtering	11 μ s	1.7	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
43 map vs. vector for dense int keys	4 μ s	2.5	○	○ -0.3	○ 0.0	○ 0.0	○ 0.0
44 Column-major matrix traversal	227 ms	0.7	○	○ -0.1	○ -2.8	○ 0.0	○ 0.0
45 std::function overhead	23 ms	0.2	●	● 0.0	○ 0.2	○ 0.2	● 0.0
46 Modifying front of vector in loop	8 μ s	2.4	○	○ 0.0	○ 0.0	○ 0.0	○ 0.0
47 Linear search on sorted data	3 μ s	0.4	●	● 0.0	● 0.2	○ 0.4	● 0.0
48 Recompiling regex repeatedly	13 μ s	0.5	●	● 0.0	● 0.0	○ 0.5	● 0.0
49 std::any when type is known	2 μ s	0.8	○	○ -0.1	○ 0.0	○ 0.0	○ 0.0
50 Fibonacci without memoization	825 ns	5.3	●	○ 0.0	● 0.0	● 0.0	● 0.0

Table 15: Copilot optimization results for Test File 2 (benchmarks compiled with -O3)

Eigenständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit selbstständig verfasst zu haben und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt zu haben.

(Ort, Datum der Abgabe)

(Unterschrift Philip Obi)