Universität Rostock

Traditio et Innovatio

# Visualization of the “Living Architecture” of software systems

## A tool for bridging the gap between static architecture and dynamic behavior


| | |
|---|---|
| Name: | Muhamad Miari |
| Student ID: | 218205391 |
| Submission Date: | 02.03.2026 |

| | |
|---|---|
| First Supervisor: | Prof. Dr. rer. nat. Regina Hebig<br>University of Rostock<br>Institute of Computer Science |
| Second Supervisor: | Dr.-Ing. Helge Parzyjegla<br>University of Rostock<br>Institute of Computer Science |


**Bachelor Thesis**
**Institute of Computer Science**

## Abstract

Architectural drift often separates static design from actual runtime behavior. This thesis introduces *Living Architecture*, an automated three-stage pipeline designed to bridge this gap by extracting high-level interactions from execution traces. The methodology employs a sequential filtering process: path-based and utilityhood filtering ($fan\_in \geq 3$, $fan\_out < 2$), adaptive dynamic depth-limiting, and a novel class-and-file boundary verification that isolates inter-component interactions from internal implementation logic.

The pipeline was validated across three distinct paradigms: procedural, MVC (Django), and event-driven (Home Assistant). Experimental results show noise reduction rates between 87.5% and 98.4% with zero false positives, ensuring all architecturally significant events were preserved. These findings demonstrate that fixed structural thresholds can generalize across diverse software architectures, providing a scalable and objective foundation for automated architectural recovery.

# Contents

# List of Figures

# List of Tables

# 1 Introduction

This chapter presents an overview of the challenges in modern software comprehension and the need to capture a system's Living Architecture. In the context of this work, Living Architecture is defined as the actual chronological order in which component classes and methods become active and communicate with each other during a specific user-triggered execution. It addresses the problem of trace-induced information overload. This chapter goes on to outline the research objectives, emphasizing the primary contribution of this thesis: an automated visualization tool that extracts and represents the Living Architecture of a software system at a specific runtime.

Program comprehension represents a fundamental aspect of software maintenance activities: before any corrective or adaptive action can be performed, the engineer must develop a sufficiently deep understanding of the system under study [6]. Through this process, a conceptual model is gradually constructed that connects the system's high-level architectural intent to its concrete implementation in source code [3],[5].

Several approaches have been proposed to support the knowledge acquisition process for software systems. On the one hand, static analysis operates on artifacts such as source code and documentation and, in principle, can account for all possible execution paths within a program [2], [31], [20]. On the other hand, dynamic analysis examines the observable behavior of a system during execution, enabling the identification of object identities and the resolution of late binding that cannot be determined solely through static means [6], [2].

Static analysis has different shapes. While a person can, for instance, read static code to see how a program was built. Nevertheless, it is much harder to discern the Living Architecture. Consequently, the so-called representation gap between the static structure of a system and its dynamic behavior is significant [7]. This gap exists because the source code, while defining the potential logic of a system, cannot visualize the actual sequence of events that occur during execution [7]. For a developer, looking at a static class definition does not reveal which methods will become active or in what specific order they will interact when a user triggers a real-world action [7]. On the other hand, dynamic analysis provides a real-time glimpse of living architecture. This behavior is only observable while the program is running. Yet, it is often buried under thousands of irrelevant background tasks, making the architectural reconstruction nearly impossible without automated assistance [22], [9], [28].

In modern software environments, when a user performs a specific action, such as clicking a register button, initiating account registration, or sending a message, the program executes a complex, multi-step chain of tasks in the background. This results in an execution trace, defined as a high-volume record of the system's internal runtime events [7], [13].

However, these logs are usually too large for a human to read or comprehend effec-

tively [15]. According to [11], this is described as the information overload problem, in which the sheer volume of execution data obscures the software's actual architectural intent. To address this problem, this thesis introduces an automated tool designed to bridge the gap through a three-pipeline system: Extraction, which records the background activity, Refinement, which separates meaningful architectural events from background technical tasks, and Visualization, which renders the results into formal PlantUML sequence diagrams. The core of this work lies in a cumulative four-stage filtering methodology designed to distinguish between essential architectural actions and non-essential infrastructure activities. This methodology operates as a progressive abstraction mechanism, systematically eliminating low-level implementation details until only the architectural components relevant to the Living Architecture are preserved. By automating this extraction, the tool enables direct observation of the program's runtime logic, providing a transparent view of the system's dynamic behavior.

## 1.1 Motivation

The primary motivation for this research is to achieve Runtime Transparency. In this work, Runtime Transparency is defined as the verifiable property of an execution trace reduction pipeline whereby all architecturally significant cross-component interactions are preserved with zero information loss while infrastructure noise is systematically eliminated to produce a clear, readable representation of the system's Living Architecture during a user-triggered execution scenario by making the live side of a program easier to observe and understand for humans. This tool is helpful and valuable for several reasons, both in academic study and professional practice.

First, one practical benefit of the proposed tool is its capacity to reduce the initial barrier to understanding for software engineers working with an unfamiliar system. Gaining sufficient comprehension of a software system is a notoriously resource-intensive process [8]. Furthermore, the authors estimate that as much as half to sixty percent of the total software engineering effort is devoted to this activity alone [3]. A primary contributor to this difficulty is the scale of data generated through dynamic analysis: even a single execution of a moderately complex program can yield traces comprising hundreds of thousands of method calls and interactions, rendering unassisted navigation of such data a substantial challenge [14], [8], [33]. Established visualization approaches, including UML sequence diagrams, are ill-suited to address this challenge at scale, as an increasing number of entities and interactions necessitates extensive two-dimensional scrolling, which disrupts rather than supports the comprehension process [8], [7]. The tool introduced in this work addresses this limitation by enabling a practitioner to invoke a specific system action and directly observe a structured representation of the corresponding runtime interactions, thereby establishing a traceable mapping between execution behavior and source code, a process term, and feature location [8]. Through this mechanism, the need to manually trace execution paths across thousands of lines of static code is eliminated, allowing new users to construct an accurate understanding of system behavior in a substantially more efficient manner.

Second, it enables faster audience adaptation, such as developers or re-engineers [8].

Companies often need their developers or employees to understand a system quickly so they can adapt and start working on it effectively [8], [11]. Understandable that recovering the living architecture helps them understand the system deeply enough in a clear way. Instead of spending weeks reading through complex code, software developers or software engineers can use this tool to see the interactions that happen in a system after they trigger an action. In [8], authors argue that seeing how a system actually behaves is the best way for a person to create a correct mental map of how that system is organized, which is essential for maintaining an accurate understanding of evolving systems.

Finally, such a tool is vital for software maintenance and error detection. Precision in maintenance requires an exact understanding of component coupling during specific tasks [1], [9]. When a system exhibits a fault or requires updating, a developer must identify precisely which classes and methods were active during the execution [6]. By automatically isolating these relevant background interactions, the tool resolves the ambiguity often found in static analysis. This enables precise error identification and allows developers to observe the actual impact of their modifications. Consequently, the maintenance process shifts from a reliance on prediction to an analysis based on observed runtime evidence, ensuring that the system remains stable as its complexity increases [26], [1].

## 1.2 Contribution

In the context of this thesis, the research question is defined as follows: How can progressive filtering techniques applied to dynamic analysis mitigate trace-induced information overload and bridge the representation gap to accurately extract and visualize a software system's Living Architecture?

Following several approaches regarding runtime filtering techniques in [23], [13], [6], [25], [30], this study implements a cumulative filtering engine targeting distinct categories of data to isolate meaningful architectural signals from the background noise. The implementation is divided into four specific stages within the filtering pipeline. Finally, this thesis evaluates how well the proposed multiple-stage filter isolates the key aspects of a program's behavior.

To achieve this, the following proposed method processes the raw execution trace through a sequential abstraction pipeline. This pipeline systematically applies increasingly restrictive criteria to discard infrastructure noise, refining the execution data across the following four distinct stages:
Stage 1: removes utility patterns and background tasks that belong to the computer's internal libraries or the framework rather than the actual program logic [13].

Stage 2: involves a depth-based filtering mechanism that stops the tool from looking at tiny technical details that are too deep to be part of the main architecture. As discussed [6], [7], this prevents the capture of low-level implementation details that are not relevant to high-level architectural understanding.

Stage 3: performs a cross-component interaction check that suppresses internal self-calls. It removes internal chatter within a single file and keeps only the meaningful interactions that cross different parts of the program [25].

Stage 4: uses classification, which is a layer-labeling. By using property inspection, the tool identifies and labels architectural layers, such as database models or controllers, giving the final diagram helpful and meaningful classification.

The proposed work introduces an automated, multi-stage filtering pipeline that synthesizes abstraction techniques from Hamou-Lhadj, Cornelissen, and Reiss into a fixed sequential composition [13, 6, 25]. Unlike prior user-driven or independent methods [16, 7, 8, 13], the proposed solution integrates utility removal, dynamic depth-limiting, and boundary checking into a single filter operating without user intervention. This staged approach facilitates modular evaluation and iterative refinement [15], ensuring architectural abstraction is achieved through a prescribed analytical path. For more details on the specific pipeline stages, refer to Section **??**.

## 1.3 Structure of the Thesis

The remainder of this thesis is structured to guide the reader from the theoretical disconnect between static code and dynamic reality toward a verified technical solution for capturing the living architecture of software. Chapter 2 introduces relevant related work and theoretical background, differentiates, among other things, between static and dynamic analysis, details Python's internal mechanics, and identifies the research gap within the current state of the art and existing profiling tools. Chapter 3 introduces the "Extraction-Filter-Visualization" pipeline, defining the strategic design of the Tracer, Filter, and PlantUML Visualizer, and establishing the project's scope within Python-based ecosystems. Chapter 4 transitions into implementation, detailing the specific logic of `tracer.py`, `filter.py`, and `visualizer.py`, with a dedicated focus on the technical management of large-scale systems through cumulative filtering and stack depth limitations. Chapter 5 presents a rigorous evaluation methodology, moving from controlled validations of different Python framework applications to real-world stress tests on complex systems such as Home Assistant and Zulip to observe performance and outcome quality. Chapter 6 provides a critical discussion of these results, comparing the tool's effectiveness against existing solutions while addressing the generalizability and threats to the validity of the chosen method, and providing an outlook for future advancements in architectural transparency. Finally, Chapter 7 draws conclusions, addresses the research question, and underscores the study's significance.

# 2 Theoretical Background

This chapter introduces the theoretical foundations for the development of a tool designed to visualize the Living Architecture of software systems. It defines key concepts such as trace extraction, filtering, and visualization, contrasting static and dynamic analysis approaches, and reviewing existing methods in the field of program comprehension.

## 2.1 Definitions and Core Concepts

The following sections define the terminology and conceptual framework used throughout this thesis, grounded in established literature on software engineering and dynamic analysis.

### 2.1.1 Living Architecture

In the context of this thesis, the term Living Architecture is used to describe the dynamic, runtime behavior of a software system as it responds to user interactions. While the specific term Living Architecture is not standard in the provided literature, it serves as a metaphor for what is formally described as the dynamic behavior or runtime architecture of a system [16].

Software architecture is typically defined as the high-level organization of a system, including its components and their interactions. However, static representations such as class diagrams often fail to capture the temporal and behavioral complexities of object-oriented systems, particularly regarding polymorphism, late binding, and object identity [8]. As noted by Cornelissen, understanding a system's behavior involves bridging the gap between high-level concepts and the source code [8].

Therefore, the Living Architecture is adopted here to represent the actual, executing structure of the software, like specific classes, methods, and objects that become active and interact during a specific usage scenario. This aligns with the concept of architectural localization, in which specific functionalities are mapped to their implementations at execution [18]. Behavioral design models focus on the causal sequences of responsibilities rather than just static dependencies [16]. By visualizing the so-called "Living" aspect, the proposed tool aims to provide an accurate picture of the system's inner workings that static analysis alone cannot reveal.

### 2.1.2 Terminology

To capture the Living Architecture, one must record the system's execution. This process is defined through the concepts of traces and tracing.

**Execution Trace:** An execution trace is defined as a chronological sequence of execution events recorded during the run-time of a program [16]. Formally, a trace $t$ can be viewed as a sequence of events where each event $e$ represents a specific change in the program state, such as a function call, a method entry, or a method exit. These events typically include unique identifiers of the event, such as method entries and exits, object creations, time stamps, and interactions between components [15], [8].

**Tracing:** Tracing is the technique of collecting these events. It involves monitoring the system while it executes a specific action or sequence of actions. This is essential since in object-oriented systems, the mere code structure does not explicitly reveal the order, extent, and frequency of execution due to dynamic dispatch [15].

**Event:** An event is defined as a distinguishable, discrete unit occurring within the execution of a program [18]. Examples include method calls, function returns, object creations, or data references, etc.

**Interaction:** Program events do not occur independently; rather, each event takes place within the scope of one or more program entities, referred to as actors (e.g., functions, objects, etc.). More precisely, an actor is defined as any syntactically identifiable code snippet within the program's source code, e.g., a function. The combination of an event and its associated actors constitutes an interaction, which can be formally understood as the relation between the event and the actors involved. For example, a function call event typically establishes a relationship between two actors: the *caller* and the *callee* [18].

**Scenarios:** To manage the inherent complexity of execution traces, program behavior is commonly examined through the concept of scenarios. A usage scenario is a single execution of the program under a specific set of inputs. One execution can yield one or more interaction scenarios. An interaction scenario is an ordered sequence of interactions whose order corresponds to the temporal sequence of the underlying events. The events included in an interaction scenario do not need to be contiguous in the raw event trace. [18].

**Techniques for Tracing:** The literature identifies several techniques for implementing a tracer, which is the component responsible for recording events:

1. **Source Code Instrumentation:** This technique involves inserting probes or logging statements directly into the source code at locations of interest, such as method entry and exit points. While effective, it requires modifying the source code and recompiling [13].

2. **Aspect-Oriented Programming (AOP):** AOP allows for the modularization of cross-cutting concerns, such as tracing. As demonstrated by Cornelissen et al. [7] in the context of visualizing test suites, AOP (e.g., AspectJ) can be used to inject tracing logic into bytecode without modifying the source code directly. This satisfies the requirement of "obliviousness," in which the system under analysis does not need manual alteration.

**Trace Collection Techniques** Several techniques have been established in the literature for implementing a tracer.

1. Source code instrumentation is one of the most common approaches, in which probes or logging statements are inserted directly into the source code at relevant locations, such as the entry and exit points of methods. Although this technique is straightforward and effective, it demands direct modification of the source code and subsequent recompilation of the system under analysis [13].

2. A less intrusive alternative is Aspect-Oriented Programming (AOP), which enables modularization of cross-cutting concerns such as tracing. Using frameworks like AspectJ, tracing logic can be woven into the bytecode without requiring manual modifications to the source code [7]. This satisfies the property of obliviousness, whereby the system under analysis remains unaware of and unaffected by the instrumentation.

3. Finally, tracing can be realized through execution environment instrumentation, in which the runtime environment itself is controlled or instrumented to emit events of interest. For instance, the Java Virtual Machine(JVM) can be configured to generate specific execution events without any alteration to the source files [13]. A comparable mechanism in Python is the built-in *sys.settrace* function, which registers a hook invoked at every function call, return, and executed line of code, enabling fine-grained monitoring of execution frames [24], [17]. While both approaches avoid source code modification, they may introduce non-negligible runtime overhead.

### 2.1.3 Filtering

Raw execution traces are often substantial, containing millions of events even for relatively short execution scenarios. This phenomenon is commonly referred to as the information overload problem [13], [8]. Filtering addresses this challenge by selectively

retaining events of interest while discarding irrelevant implementation details and redundant information, thereby raising the level of abstraction at which the trace is represented [13], [15].

In the context of living architectural analysis, filtering plays a particularly critical role, as it enables the transformation of a low-level stream of execution events into a high-level architectural view. In the absence of adequate filtering, visualizations derived from such traces quickly become cluttered and difficult to interpret.

**Adaptive Runtime Filtering Techniques** Wagner et al. [30] propose a set of adaptive runtime filtering strategies aimed at reducing trace volume and minimizing measurement overhead. Two representative strategies are described below.

**Leaf Node Filtering** targets regions of the call tree that do not invoke any further instrumented functions, i.e., functions with no child activities. The underlying assumption is that the structural relationships within the call tree, specifically which component calls which, are generally more informative for understanding program behavior than the fine-grained computations occurring at the lowest levels of the call stack. This strategy reduces trace detail by one level of abstraction without requiring application-specific context [30].

**Similar Duration Filtering** removes repeated executions of the same function whose runtime durations are comparable to the running average. Rather than retaining every individual execution, this technique replaces groups of similar-duration occurrences with a representative summary, retaining only those executions that deviate meaningfully from the expected behavior. A tolerance parameter is additionally employed to prevent the inadvertent removal of executions that exceed the average only marginally [30].

**Utilityhood-Based Filtering** Hamou-Lhadj and Lethbridge propose a complementary filtering approach based on the identification of utility components, i.e., low-level implementation constructs such as data-structure helpers or generic utility functions that, while frequently invoked, contribute little to the high-level understanding of the system [13]. To quantify the likelihood of a routine being a utility, the authors define the utilityhood metric $U(r)$, which combines fan-in and fan-out information as follows:

$$U(r) = \frac{\text{Fanin}(r)}{N} \cdot \frac{\log\left(\frac{N}{\text{Fanout}(r)+1}\right)}{\log(N)} \tag{2.1}$$

where $N$ denotes the total number of routines in the call graph, routines with high fan-in (i.e., called by many other routines) and low fan-out (i.e., calling few others) receive higher utilityhood scores and are considered strong candidates for removal. By suppressing such routines, the filtered trace foregrounds the core domain-level interactions rather than low-level implementation noise. The exact threshold above which a routine is classified as a utility is context-dependent and may be adjusted according to the desired level of detail [13], [15].

**Pattern Matching** Pattern matching, as discussed by Hamou-Lhadj and Lethbridge, addresses the size explosion problem by grouping recurring sequences of events into higher-level execution patterns. Rather than presenting the analyst with every individual occurrence of a repeated interaction sequence, the trace is compacted such that each distinct behavioral pattern appears only once [13]. This concept has been adopted under various names across different tools, including behavioural patterns: Shimba, interaction patterns in ISVis, and collaboration patterns in The Collaboration Browser. When properly generalized, pattern matching can substantially reduce trace size. However, selecting appropriate matching criteria (e.g., exact matching versus interleaved matching) remains a non-trivial challenge, as different criteria yield different levels of trace compaction and may affect the analyst's ability to interpret the underlying system behavior correctly [13].

### 2.1.4 Visualization of the Living Architecture

Visualization serves as the primary interface between the large volumes of data generated by dynamic analysis and the human analyst's cognitive processes. As demonstrated by Grundy and Hosking, graphical representations of running system information provide developers with a means to observe and reason about complex software behavior that cannot be readily inferred from static artifacts alone [12]. In the context of Living Architectural analysis, the goal of visualization is to support the construction of an accurate conceptual model of the system's components and the runtime relationships that realize specific functionalities. A central challenge in this regard is designing visualizations that remain interpretable and meaningful as both the system's complexity and the volume of the underlying execution trace increase [12].

#### Visualization Techniques for Execution Traces

1. UML sequence diagrams, also referred to as message sequence charts, represent one of the most widely used techniques for visualizing the chronological flow of interactions between system entities [8]. In this representation, actors are arranged horizontally while time progresses along the vertical axis, making the ordering of interactions intuitive and easy to follow [16], [8]. As demonstrated by Souder and Mancoridis, reconstructing sequence diagrams from execution traces provides a meaningful basis for design recovery and supports the decomposition of system functionality into comprehensible behavioral slices. This technique effectively bridges the gap between low-level runtime events and higher-level design models by making the order of execution and the identities of participating objects explicit [29]. However, sequence diagrams are subject to well-known scalability limitations. As noted in [8] by the authors, these diagrams rapidly become difficult to navigate when the number of interacting components or the volume of events grows, often necessitating two-dimensional scrolling that hinders rather than supports the comprehension process.

2. **Information Murals**
   To address the scalability limitations of standard sequence diagrams when deal-

ing with traces containing millions of events, the Information Mural technique was proposed by Jerding and Stasko [18], [19]. This approach constructs a two-dimensional miniature representation of the entire information space, scaled to fit within the available screen area. As implemented in the Extravis tool by Cornelissen et al., interactions are rendered as color-coded rectangles along the vertical axis, with the system's full hierarchical structure projected onto the horizontal axis. This massive sequence view allows analysts to identify global execution phases, recurring behavioral patterns, and structural outliers that would remain invisible in a fine-grained, scrollable sequence diagram.

3. **Circular Bundle Views**
   To simultaneously convey structural coupling and dynamic behavior, Cornelissen proposes the circular bundle view, in which a system's structural entities (e.g., classes and packages) are projected onto the circumference of a circle, with dynamic call relationships rendered as bundled splines through the circle's interior [8]. The visual bundling of edges that share common parent components substantially reduces visual clutter and makes high-level architectural communication patterns, such as intensive interaction between specific subsystems, immediately apparent. Furthermore, the thickness of each bundled edge encodes the frequency of calls between two entities, thereby providing an intuitive measure of coupling strength [8]. Used in conjunction with the massive sequence view, the circular bundle view forms a synergistic pair of linked perspectives that supports navigation across multiple levels of granularity within a single execution trace [8], [6].

**Automated Generation of Behavioral Models from Execution Traces** The selection of an appropriate visualization mechanism in this work is driven by the need to produce behavioral models automatically from execution traces that have undergone a filtering process [13]. Prior research has demonstrated that constructing such models by hand becomes impractical as system complexity grows, which underscores the necessity of automated approaches capable of translating filtered trace data directly into visual representations. UML sequence diagrams have been recognized as a particularly suitable output format in this context, given their established role in depicting the runtime behavior of software systems [13], [15].

A key principle underlying this process is the separation of concerns between two operationally distinct stages. The first stage is concerned with reducing the raw trace to a manageable and meaningful subset, achieved through the systematic elimination of low-level implementation constructs that do not contribute to a high-level understanding of the system. The second stage involves rendering the resulting reduced trace in a form that can be readily interpreted by software engineers [15]. This two-stage structure supports an iterative workflow in which the output is progressively refined until the desired level of abstraction is achieved. Notably, the choice of visual notation in the second stage is independent of the summarization process itself. UML sequence diagrams are adopted on pragmatic grounds, as (a) they constitute a broadly accepted standard for behavioral modeling, and (b) the correspondence between trace elements

and their sequence diagram counterparts can be established in a systematic and largely straightforward manner [15], [13].

To address the visualization requirements outlined above, a text-to-model transformation approach is adopted, in which filtered execution events are encoded in a domain-specific language (DSL) that describes the constituent elements of the target diagram. A rendering engine subsequently processes this textual representation to produce the final graphical output. This approach conforms to the design requirements defined by Cornelissen et al., namely: [7] obliviousness, which stipulates that the system under analysis requires no modifications to its implementation to support examination, and [7] extensibility, which requires that the framework support the addition of customized event listeners, abstraction and filtering mechanisms, and output renderings to enable flexible visualization and integration within external environments.

In the present work, PlantUML is employed as the script-driven text-to-diagram tool, producing output that conforms to the UML notation standard. This choice is justified on the following grounds: (a) the transformation from a filtered trace representation (e.g., JSON) to the visualization syntax can be fully automated without human intervention, supporting rapid feedback during trace exploration, (b) the use of UML as the output notation ensures that the resulting behavioral model is interpretable by software engineers familiar with standard design artifacts, thereby reducing associated cognitive overhead, and (c) the approach structurally decouples the trace processing logic from the rendering logic, ensuring that the visualization step is strictly concerned with the presentation of an already summarized trace, while the underlying analysis components remain focused on metric-based filtering and pattern detection [13].

## 2.2 Static vs. Dynamic Analysis

Two primary methodologies are employed for analyzing and understanding software systems: static analysis and dynamic analysis.

Static analysis examines a system's artifacts, such as source code and documentation, without requiring program execution. In principle, this approach can account for all possible execution paths within the system and is well-suited to determining structural properties. However, static analysis faces inherent limitations in object-oriented systems because mechanisms such as polymorphism and late binding hinder the precise identification of the method implementation that will be invoked at runtime based solely on source code inspection[8]. Moreover, existing static modeling tools frequently operate at a low level of abstraction and fail to provide a sufficiently broad range of architectural views, resulting in representations that do not adequately reflect the system's runtime complexity [12].

Dynamic analysis, by contrast, operates on data collected from an executing program. This approach is capable.

# 3 Proposed Solution: Living Architecture

The development of the proposed solution is motivated by a fundamental gap identified in the existing literature: while source code analysis tools exist for a variety of programming languages (e.g., C, C++, and Java) and provide researchers with infrastructure for automatic modularization, design extraction, and metrics, source code analysis alone does not provide sufficient information to understand a program completely, because there are components and relations that only manifest during runtime [29]. This limitation establishes the need for a dynamic analysis approach capable of capturing and presenting the system's Living Architecture.

To address this gap, a three-stage pipeline is proposed, structured around the principle of Extract → Filter → Visualize. In this work, the solution is referred to as Living Architecture. This architectural model was inspired by Cornelissen's work on dynamic analysis tools [9]. The proposed solution buildsefines it, and separates the concerns of data acquisition, data processing, and data presentation, allowing each on that work, r component to evolve independently through its own modifications and other solutions [29].

To systematically design and evaluate this system, it is necessary to explicitly differentiate between the system architecture and the data reduction logic:
1. The Three-Stage Pipeline: This is the overarching architecture of the proposed solution. It governs the entire lifecycle of the dynamic analysis process, divided into three decoupled components: Extract, Filter, and Visualize [9].
2. The Four-Stage Filtering Concept: This is logic executed exclusively within the second component (the Filter) of the pipeline. It consists of four distinct, sequential algorithms designed to progressively eliminate noise from the extracted data by the tracer.

A simple Business Process Model and Notation (BPMN) diagram was selected to illustrate the proposed solution. [32]. Figure 3.1 illustrates the proposed solution. The pipeline automates architectural visualization by extracting runtime execution data via `sys.settrace` upon a user trigger. The workflow follows a sequential analytical path: Stage 1a filters utilities by dropping `lib/python` calls, while Stage 1b calculates utilityhood metrics via fan-in/fan-out analysis; Stage 2 applies dynamic depth-limiting; Stage 3 executes architectural boundary checks; and Stage 4 performs layer classification (UI, Logic, Data).

The final output of the pipeline is rendered as a PlantUML sequence diagram, a script-driven text-to-diagram format that conforms to the UML notation standard and supports fully automated generation from filtered trace data, as prescribed by Cornelis-

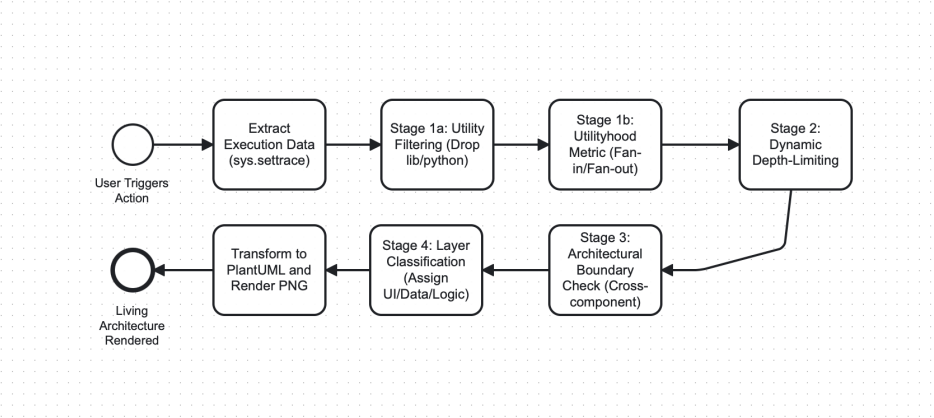



Figure 3.1: BPMN diagram representing the sequential workflow stages.

sen for extensible dynamic analysis frameworks [7], [21].

Python was selected as the implementation language for the proposed pipeline for two distinct reasons. First, the dominant body of prior dynamic analysis research has concentrated on languages such as Java and C++, leaving Python comparatively underexplored as a subject of dynamic architectural analysis. Second, and of greater technical significance, Python provides a built-in, interpreter-level tracing mechanism: the `sys.settrace` function, which enables fine-grained monitoring of execution events at the frame level without requiring bytecode modification, source code instrumentation, or external tooling [10], [24]. This native capability makes Python uniquely suited to the kind of non-intrusive runtime observation that the proposed pipeline requires, and it satisfies the obliviousness property identified by Cornelissen [7], whereby the system under analysis can be left unmodified to support examination.

Living Architecture Components

The proposed pipeline comprises three functional components, each responsible for a distinct phase of the analysis. These three components correspond to the Trace–Filter–View model identified by Cornelissen as the canonical architectural pattern for dynamic analysis tools [9]. Their functional separation ensures that each component can be developed, tested, and refined independently, and that intermediate serialized artifacts, specifically, the raw JSON trace and the filtered JSON trace, are available for empirical inspection and per-stage evaluation at every point in the pipeline.

The Tracer: Dynamic analysis fundamentally relies on the accurate recording of method calls and system interactions during execution. Traditional external instrumentation tools often require complex modifications to the source code or the execution environment, which can alter the system's native behavior [10]. To mitigate these overheads, the tracer implemented in this thesis utilizes Python's native `sys.settrace` functionality. By hooking directly into the Python interpreter, the tracer passively captures execution events (entries, exits, and returns) generated by the user's action. This approach ensures that the raw living architecture is recorded comprehensively and accurately without manually altering the target application's source code [24].

The Filter: The following is based on the literature review as described in the The-

oretical Background chapter. Execution traces natively generate massive volumes of data; therefore, the core of trace summarization relies heavily on the filtering mechanism [15], [13]. A review of the literature indicates that applying a single filtering technique either leaves substantial noise or aggressively removes vital architectural information [6], [9]. Consequently, the most effective approach observed among authors in this domain is the combination of multiple filtering techniques.

At each intercepted call event, the Tracer captures the *caller*, the *callee*, and the source file paths and line numbers of both parties. This triple corresponds to the foundational event structure described by Reiss and Cornelissen [25], [7]. And to the trace representation used by Hamou-Lhadj [13], in which each trace event records the sender's and receiver's classes and the invoked method. Because Python uses polymorphism and duck typing, the actual runtime class of the receiving object must be resolved dynamically at capture time rather than inferred from declared types. This ensures that the trace reflects the actual living type of each participant rather than its statically declared type.

The filtering component is the intellectual core of the proposed solution.
While prior work employs multiple abstraction techniques within single tools in an interactive, user-driven manner [16], [7], no existing work prescribes a fixed, automatically applied sequential pipeline that combines utility removal, dynamic depth limiting, and architectural boundary checking into a single integrated filter operating without user intervention. The techniques available in the literature are either evaluated independently against each other or offered as user-selectable options rather than as a prescribed sequential composition[8],[13]. This gap motivates the approach taken in this work, in which concepts drawn from Hamou-Lhadj, Cornelissen , and Reiss are synthesized into one automatically applied, multi-stage pipeline [13], [6], [25].
Beyond the theoretical justification for combining techniques, there was a practical reason for organizing them into sequential stages: stages make the filter easier to understand, develop, and evaluate. After each stage, the result can be independently inspected as a visualization, and the remaining noise can be identified before deciding on the next stage. The staged structure was not designed in its entirety upfront. Instead, each stage was added after observing the noise remaining in the visualization produced by all previous stages combined, and selecting the most appropriate technique from the literature to address that specific category of noise. This was applied in a pilot application and was not used in the three applications on which the proposed solution was later applied.

Stage 1 (Utility Filtering): The initial step is to remove unnecessary information that does not reflect high-level architectural interactions. This stage removes two distinct types of noise:
Stage 1a: First, all events from Python's standard library and from installed packages, such as Django and other third-party frameworks, were removed. These modules reside at paths matching patterns such as lib/python or site-packages. They are framework and language infrastructure components, not part of the application's architecture.

Stage 1b (Utilityhood Metrics): Following the removal of external libraries in Stage 1a, the pipeline addresses internal utility noise (e.g., loggers, validators) using utilityhood metrics [16, 13]. This stage leverages the topological signature of components within the call graph: utility components function as "shared servants," characterized by **high fan-in** (called by many distinct *callers*) and **low fan-out** (calling few other components) [23]. Conversely, components with high fan-out are classified as architecturally significant orchestrators and are preserved. The filter constructs a call graph from the trace data and computes metrics for each method. A method is classified as a utility and subsequently pruned if it satisfies the condition of $fan\_in \geq 3$ and $fan\_out < 2$. This specific threshold enables the automated differentiation between cross-cutting utility functions and core business logic based on their system-wide connectivity [14].

Stage 2: A dynamic depth-limited approach was adopted and adjusted, based on the target-size variant [6]. Rather than applying a fixed cutoff, the appropriate depth is determined automatically from the distribution of events across stack-depth levels within each trace. Events are accumulated from the shallowest depths downward until a target trace size is reached, and the depth at which that target is met becomes the cutoff. This means a shallow, broad application may stop at depth 5 while a deeply nested framework may require depth 15, all determined by the same logic without any manual tuning. This adaptivity is precisely what makes the approach scalable across both small and large applications [6], [7].

Stage 3: A simple deduplication process was developed to remove duplicate call records. It was combined with a solution was found in the work of Reiss and Renieris [25]. For each remaining call, the filter checks whether it crosses an architectural boundary by examining both the *caller* and *callee's* class and file origins. A call is kept if the *caller* and *callee* reside in different classes or different files, representing a cross-component interaction. The call is dropped only if both the *callerandcallee* are of the exact same class and in the same file, representing internal implementation. The underlying principle is that architectural models describe how components interact, not how they function internally [25].

Stage 4: Rather than removing events, this stage enriches the surviving interactions by grouping all retained method calls by their class. Each class appearing in the final filtered trace becomes a named participant in the architectural view. The participants are then classified into one of three coarse-grained architectural tiers, namely Presentation Layer, Business Logic Layer, or Data Layer, through heuristic inspection of class attributes, names, and inheritance hierarchies. This provides a clean list of the system's active components. Stage 4 pursues the same goal in a form appropriate to the output of the sequence diagram. It is placed last in the pipeline so that classification effort is not spent on components that will be removed by the earlier stages.

The Visualizer: This process transforms the filtered trace into a PlantUML-encoded

UML sequence diagram, rendering the Living Architecture in a human-readable form. Two design decisions shape this component: the selection of UML sequence diagrams as the output notation, and the adoption of PlantUML as the rendering engine through a text-to-model transformation approach, so the visualizer component automatically transforms the filtered trace data into the PlantUML format, and text-to-model transformation accurately renders the verified, dynamic runtime events into a standard UML sequence diagram, providing an understandable and scientifically supported presentation of the living architecture.

# 4 Implementation

Translating the theoretical concepts established in the previous chapters into a functional software tool is a fundamental requirement of this research. Because dynamic analysis inherently relies on observing a running program in its actual environment [2], the proposed methodology must be implemented as an executable program capable of recording actions as they occur. This chapter details the technical implementation of the Living Architecture pipeline. The software was developed entirely in Python and is structured into three core modules: the tracer (`tracer.py`), the filter (`filter.py`), and the visualizer (`visualizer.py`).

## 4.1 The Tracer

Event Interception and Memory Analysis:
For each intercepted action, the tracer looks at the active "Frame" in the computer's memory. A frame can be understood as a temporary workspace or a snapshot of what the program is doing at that exact millisecond. The necessary architectural information is extracted directly from this workspace:

The implementation is specifically designed to isolate call events, which correspond to method entries. For each intercepted call, the tracer accesses the active `FrameType` object residing in memory, which holds the current execution context. The necessary architectural metadata is extracted directly from this frame:

- **Callee Identification:** The script accesses the frame's code object (`f_code`) to retrieve the active file path (`co_filename`) and method name (`co_name`). To determine the specific object-oriented component, the frame's local variables (`f_locals`) are queried. If a `self` reference is detected, the script dynamically resolves the active class instance (`instance.__class__.__name__`).

- **Caller Identification:** Establishing the behavioral sequence requires identifying the origin of the interaction. The implementation achieves this by traversing the execution stack backwards, accessing the parent frame via the `f_back` attribute. The same extraction logic is applied to the parent frame to retrieve the `caller_file`, `caller_method`, and `caller_class`.

Additionally, a `while` loop continuously traverses the `f_back` chain to calculate the absolute stack depth of the current execution point. This integer value is stored for subsequent reduction operations. The exact implementation of the Tracer is found in Appendix 8.1.

## 4.2 The Filter

The core mechanism for trace reduction is encapsulated within the `ArchitectureFilter` class. A systematic, four-stage filtering pipeline was implemented. This module is designed to sequentially process the recorded interactions, mathematically identifying and removing non-architectural noise while preserving the true behavioral sequence. More information regarding the incremental development of the Filter is found in Appendix 8.4

### 4.2.1 Stage 1

The first operation performed by the filter is the elimination of low-level software mechanics that do not contribute to the high-level architecture. This stage is divided into two distinct computational steps:

**Stage 1a:** This step is executed during the initial data capture. It is implemented via the `should_trace()` method, which checks the file path of every intercepted method against a predefined list of `ignore_patterns` (e.g., `lib/python`, `site-packages`, `unittest`). If the file path contains any of these strings, the method returns `False`, and the execution event is completely ignored. This ensures that the internal operations of external packages and standard Python libraries are excluded from the dataset, keeping the focus strictly on the target application. This step was not implemented in Hamou-Lhadj's work [13].

```
self.ignore_patterns = [
  "site-packages",
  "dist-packages",
  "lib/python",
  "<string>",
  "unittest",
  "pydev",
  "tracer.py",
  "visualizer.py",
  "filter.py",
]
```

**Stage 1b:** After the environment noise is removed, internal utility methods (such as logging functions or data formatters) still remain in the trace. To remove these, the `apply_stage1_utility_filter()` method is executed during post-processing. A mathematical representation of the interactions, known as a call graph, is constructed via the `_build_call_graph()` function. The system then calculates two specific metrics for every method: "fan-in" (the number of unique methods that call it) and "fan-out" (the number of unique methods it calls). Based on the theoretical framework established by Hamou-Lhadj [13], utility functions can be identified by their high connectivity. In this implementation, if a method exhibits a fan-in $\geq 3$ and a fan-out $\leq 2$, it is classified as a utility and programmatically removed from the trace. This is effective because

true architectural components typically delegate tasks rather than being invoked by a universally diverse set of sources. The complete code snippet of the Filter is found in Appendix 8.2.

### 4.2.2 Stage 2:

Observation of the data after Stage 1 revealed that the trace still recorded deep, repetitive execution chains. While limiting stack depth is a proven method for reducing trace size, applying a strict, fixed-depth limit is problematic. A fixed limit might be too aggressive for a small application and too lenient for a complex software framework.

To resolve this, the `apply_stage2_filtering()` method implements a dynamic depth calculation algorithm, adjusted from the concepts proposed by Cornelissen [6].The script first iterates through the trace and counts the total number of events occurring at each specific depth level (`depth_counts`). Next, it progressively accumulates these counts from the shallowest depth downward until a predefined `target_size` is reached. The depth at which this target is met becomes the calculated `max_depth`. All execution events located deeper than this dynamic threshold are subsequently removed. This implementation ensures that the filtering process scales automatically to fit the specific complexity of any analyzed system.

### 4.2.3 Stage 3:

Even after the first two stages, significant repetition often remains within the filtered data. Standard repetition removal algorithms were evaluated but found to be overly aggressive, occasionally deleting valid sequence data. To address this without losing architectural integrity, the `is_architectural_interaction()` method was developed, drawing inspiration from the trace encoding concepts presented by Reiss and Renieris.

This stage distinguishes between true architectural communication and purely internal implementation logic. For every remaining call, the script evaluates the file and the class of the sender (caller) against the receiver (callee). The logic is implemented as follows:

- If the call occurs between two completely different files, it is classified as a cross-module interaction and is preserved.
- If the call occurs within the same file but between two different classes, it is classified as a cross-component interaction and is preserved.
- If the sender and receiver share the exact same class and the exact same file, the interaction is classified as internal implementation logic and is discarded.

The underlying principle enforced by this code is that software architecture models how distinct components interact, rather than how a single component functions internally.

### 4.2.4 Stage 4:

The final stage of the filter does not remove data; rather, it structures the preserved interactions to prepare them for visualization. The `categorize_class()` method examines the properties of each remaining component to infer its high-level architectural role.

This is achieved by inspecting the objects for specific programmatic clues using Python's `hasattr()` function and string matching. For example, if a component possesses attributes like `winfo_id` or `mainloop`, or if its name contains "View", it is assigned to the `Presentation_Layer`. Instances possessing a `cursor` or `execute` attribute are mapped to the `Data_Layer`. Any component lacking these specific clues defaults to the `Logic_Layer`. This heuristic classification ensures that the final sequence diagram organizes the extracted data into a universally recognized architectural format.

## 4.3 The Visualization

The final phase of the dynamic analysis pipeline requires translating the filtered, text-based data into a graphical format. Even when heavily reduced, textual execution logs remain difficult for human readers to interpret efficiently [7]. To understand the sequence of events and the overall system behavior, the data must be presented as a behavioral design model [16]. This transformation is handled by the `PlantUMLGenerator` class within the `visualizer.py` file. Its primary technical objective is to automatically convert the serialized JSON trace data into a standardized Unified Modeling Language (UML) sequence diagram.

To ensure the final diagram is both accurate and visually accessible to stakeholders, the visualization engine executes a structured, multi-step process: data ingestion, final verification, participant extraction, syntax generation, and image rendering. The complete implementation of the Visualization is found in Appendix 8.3.

### 4.3.1 Data Ingestion and Final Post-Processing

Before drawing the architecture, the visualizer loads the `trace_data.json` file generated by the tracer. To guarantee that the diagram only displays the most refined data, the `generate()` method first subjects the loaded data to a final round of post-processing using the `ArchitectureFilter` engine.

Specifically, it applies Stage 1b (the fan-in/fan-out utility filter) to mathematically remove highly connected helper methods. Following this, it initiates Stage 3 (the architectural boundary check). The script iterates through the trace log to verify whether the interactions represent true cross-component communication. If an event passes these final mathematical and structural checks, it is appended to an `architectural_events` array. This late-stage verification acts as a final safeguard, ensuring that the resulting diagram strictly models high-level architecture rather than low-level implementation logic.

### 4.3.2 The Two-Pass Extraction Algorithm

Once the dataset is finalized, the script must determine what to draw. This is achieved through a two-pass algorithm that scans the data array.

**Pass 1: Identifying the Participants.** In a sequence diagram, the entities that communicate with each other are called "participants". The script scans every event to identify the sender (caller) and the receiver (callee). To make the diagram logically coherent, the code normalizes varying entry points (such as `Script_Controller` or `External_User`) into a single, unified actor named `User_Action`. Every unique component discovered during this scan is saved into a specific memory set, ensuring that no component is drawn twice.

**Pass 2: Building the Sequence.** The script scans the data a second time to map the chronological flow of the program. For every interaction, it extracts the sender, the receiver, and the specific task (method) performed. This data is then formatted into a strict, text-based directional arrow syntax required by PlantUML (e.g., `caller -> callee : method()`). Interactions that only involve wrapper scripts (like `Global_Script`) are explicitly skipped to prevent visual clutter.

### 4.3.3 Syntax Generation and Rendering

The text-based instructions collected during the two-pass algorithm must be assembled into a complete script. The `_write_puml()` method creates a new text file and injects standard formatting headers (`@startuml`). To ensure the output is professional and readable, strict styling parameters are applied, such as center alignment and specific padding between elements.

A system diagram must clearly distinguish between the human user triggering the system and the internal software components doing the work [16]. The script enforces this by explicitly defining the `User_Action` as an `actor` (drawn as a stick figure) on the far left of the diagram, while all other system components are defined as `participant`s (drawn as rectangular boxes) to the right. A scientific legend is also automatically appended to the bottom of the file to explain these visual elements to the reader.

Finally, the `_render_png()` method attempts to convert the text-based script into a graphical image. It utilizes the `plantuml` Python library to establish a connection with a remote PlantUML server over the internet. The script transmits the text file, and the server processes the syntax, drawing the actual visual elements. If successful, the server returns a fully rendered `.png` image file, which is saved locally. This automated transformation bridges the gap between raw machine execution and human comprehension, successfully visualizing the system's "Living Architecture" [7].

# 5 Results and Evaluation

## 5.1 Results

In the following section, the three applications on which the proposed solution was applied will be briefly introduced. Moreover, the results of each application will be presented.

### 5.1.1 Application 1:

`console_app_1 (Sorter Application)` or The Sorter Application is a self-authored Python application that implements a merge sort algorithm alongside supporting components. Partially developed during an academic seminar, it was selected for its familiarity with its codebase and controlled test-case. Furthermore, Application 1 generates the specific types of noise targeted by the pipeline and exhibits clear architectural interactions. The decision to begin with a familiar, self-authored application reflects established practice in dynamic analysis research: a controlled environment is required to confirm the correctness of a tracer and filter before applying them to external systems with undocumented or complex internal structures. The main components are:

- AppController: Entry point that coordinates all other components.
- DataGenerator: Generates random integer arrays using Python's `random` module.
- MergeSorter: Implements divide-and-conquer merge sort with recursive `sort` and `_merge` methods.
- Statistics: Computes averages and min/max values on sorted data.
- Logger: A cross-cutting utility called from five different methods to record application events.
- Validator: A cross-cutting utility called from three different methods to validate inputs.

The application does not require external user input data, as it internally generates a random list of six integers between 1 and 100 to use as the target dataset. It processes this generated list through a merge sort algorithm and a statistics calculator. The final outputs printed to the console include the sorted list, the average value, and the minimum and maximum values of the array.

The raw trace captures 72 events over the execution of one complete sort-and-validate job. The pipeline is configured with a fan-in threshold of $T_U = 3$, a fan-out threshold of $T_{FO} = 2$, and nine filename exclusion patterns covering standard library locations.

### Pre-filtering

The pre-filtering analysis retained all 72 initial events, yielding a 0.00% reduction across the libraries, depth, and boundary evaluation stages. Consequently, the entire dataset was preserved and classified exclusively within the participant layer. This unreduced dataset establishes the baseline reference point for all subsequent filtering stages.

Table 5.1: stage 0 (pre-filtering)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 72 |
| Events kept: | | 72 |
| Reduction: | | 0.00% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 0 | N/A |
| Stage 2 (Depth) | 0 | (post-processed) |
| Stage 3 (Boundary) | 0 | N/A |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 72 | Participant: 72 |

### Stage 1

Activation of the Stage 1 filter resulted in a 17.81% reduction of the dataset, isolating and removing 13 utility and library events from the 73 total events analyzed. Specifically, external library calls corresponding to random number generation functions were eliminated, while the remaining 60 events were preserved exclusively within the participant layer.

Table 5.2: stage 1 (after activating stage 1)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 73 |
| Events kept: | | 60 |
| Reduction: | | 17.81% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 13 | randint, _randbelow_with_getrandbits, randint... |
| Stage 2 (Depth) | 0 | (post-processed) |
| Stage 3 (Boundary) | 0 | N/A |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 60 | Participant: 60 |

### Stage 2 results

The concurrent activation of Stage 1 and Stage 2 filters reduced the dataset by 50.68%, retaining 36 of the original 73 events. This cumulative reduction was achieved by eliminating 13 library and utility events in Stage 1, followed by the targeted removal of 24 deep events in Stage 2 based on a calculated maximum depth threshold of 6.

Table 5.3: stage 2 (after activating stage 1 and 2)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 73 |
| Events kept: | | 36 |
| Reduction: | | 50.68% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 13 | randint, _randbelow_with_getrandbits, randint... |
| Stage 2 (Depth) | 24 | (post-processed) |
| Stage 3 (Boundary) | 0 | N/A |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 36 | Participant: 60 |

### Stage 3 results

The sequential application of the first three filtering stages yielded a cumulative dataset reduction of 63.01%, retaining 27 of the initial 73 analyzed events. Specifically, the activation of the Stage 3 architectural boundary filter isolated and removed 9 internal calls

by filtering for cross-file and cross-class interactions. This progressive refinement eliminated structural noise, leaving a concentrated subset of essential events for subsequent layer distribution analysis.

Table 5.4: stage 3 (after activating stage 1,2 and 3)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 73 |
| Events kept: | | 27 |
| Reduction: | | 63.01% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 13 | randint, _randbelow_with_getrandbits, randint... |
| Stage 2 (Depth) | 24 | (post-processed) |
| Stage 3 (Boundary) | 9 | 9 internal calls |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 27 | Participant: 60 |

### Stage 4 results

The completion of the final Stage 4 filter maintained the cumulative 63.01% reduction, yielding a final dataset of 27 retained events from the original 73. During this stage, the system performed layer distribution analysis on the remaining subset, attributing the architectural interactions to the logic layer with a reported breakdown count of 60. This final classification isolates the core functional components of the application following the systematic removal of library, depth, and internal boundary noise.

Table 5.5: stage 4

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 73 |
| Events kept: | | 27 |
| Reduction: | | 63.01% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 13 | randint, _randbelow_with_getrandbits, randint... |
| Stage 2 (Depth) | 24 | (post-processed) |
| Stage 3 (Boundary) | 9 | 9 internal calls |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 27 | Logic_Layer: 60 |

### 5.1.2 Application 2:

The second application is an external system, the open-source Django REST Framework (DRF) Example App, cloned from the "gothinkster/productionready-django-api" repository. This Application provides RESTful APIs; it expects JSON-formatted HTTP requests as input to process standard CRUD operations and returns corresponding structured JSON responses. This application was selected because it strictly adheres to real-world applications and provides an API specification, providing a test-ready architecture rather than a simplified, self-authored test script. The code incorporates complex internal mechanisms, nested serializers, and authentication logic, which naturally generate a high volume of structural and utility noise that the proposed pipeline targets.

#### Pre-filtering

The baseline pre-filtering analysis captured 500 events with a 0.00% reduction across all initial stages. These retained events were categorized into 320 participant-layer events and 180 script-layer events. This unreduced dataset serves as the quantitative reference point for subsequent filtering evaluations.

Table 5.6: stage 0 (pre-filtering)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 500 |
| Events kept: | | 500 |
| Reduction: | | 0.00% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 0 | N/A |
| Stage 2 (Depth) | 0 | (post-processed) |
| Stage 3 (Boundary) | 0 | N/A |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 500 | Participant: 320, Script: 180 |

#### Stage 1

Activation of the Stage 1 filter achieved a 98.20% dataset reduction by eliminating 491 library and utility events from the 500-event baseline. The 9 retained events were subsequently classified into 5 script layer events and 4 participant layer events.

Table 5.7: stage 1 (after activating stage 1)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 500 |
| Events kept: | | 9 |
| Reduction: | | 98.20% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 491 | get, get, generic... |
| Stage 2 (Depth) | 0 | (post-processed) |
| Stage 3 (Boundary) | 0 | N/A |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 9 | Script: 5, Participant: 4 |

### Stage 2

The combined activation of Stage 1 and Stage 2 filters maintained a 98.20% reduction, with a calculated maximum call depth of 27 triggering no additional removals. The dataset remained at 9 events, preserving the distribution of 5 script and 4 participant layer events. This indicates all post-Stage 1 events resided within the permissible architectural depth for this trace.

Table 5.8: stage 2 (after activating stage 1 and 2)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 500 |
| Events kept: | | 9 |
| Reduction: | | 98.20% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 491 | get, get, generic... |
| Stage 2 (Depth) | 0 | (post-processed) |
| Stage 3 (Boundary) | 0 | N/A |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 9 | Script: 5, Participant: 4 |

### Stage 3

The concurrent activation of the first three filtering stages achieved a 98.40% cumulative reduction, retaining 8 of the initial 500 events. Stage 3 specifically eliminated one internal call by enforcing architectural boundary constraints. The final subset concentrated the trace into a distribution of 5 script and 4 participant layer events.

Table 5.9: stage 3 (after activating stage 1, 2 and 3 )

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 500 |
| Events kept: | | 8 |
| Reduction: | | 98.40% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 491 | get, get, generic... |
| Stage 2 (Depth) | 0 | (post-processed) |
| Stage 3 (Boundary) | 1 | 1 internal calls |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 8 | Script: 5, Participant: 4 |

#### Stage 4

The comprehensive activation of all filtering stages yielded a cumulative reduction of 98.40%, retaining 8 of the original 500 events. The final dataset was categorized into 5 script and 4 logic-layer events. This demonstrates the pipeline's efficacy in isolating core application logic from significant architectural noise.

Table 5.10: stage 4 (after activating all stages )

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 500 |
| Events kept: | | 8 |
| Reduction: | | 98.40% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 491 | get, get, generic... |
| Stage 2 (Depth) | 0 | (post-processed) |
| Stage 3 (Boundary) | 1 | 1 internal calls |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 8 | Script: 5, Logic_Layer: 4 |

### 5.1.3 Application 3:

Home Assistant, an open-source home automation platform acquired from the official "Open Home Foundation" repository. The selection of this application was motivated by the need for a widely used one, as Home Assistant is a ubiquitous IoT ecosystem. It processes inputs from diverse integrations, such as Amazon Alexa and Philips Hue, to generate automated control signals and state-based dashboard outputs. This

production-grade codebase was identified as an optimal match for research objectives because of its complex, event-driven architecture and reliance on asynchronous background loops.
By utilizing this external application rather than a containerized environment like Docker, the tracing tool directly interfaces with the native Python call stack. This setup provides a rigorous use-case for testing different aspects of the filtering pipeline.

### Pre-filtering

The baseline pre-filtering analysis for the Home Assistant application captured 951 total events, resulting in a 0.00% reduction across all initial stages. These events were categorized into 283 script-layer interactions and 668 participant-layer interactions. This unreduced dataset serves as the quantitative reference point for assessing the efficacy of the filtering pipeline on the modular IoT architecture.

Table 5.11: stage 0 (pre-filtering)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 951 |
| Events kept: | | 951 |
| Reduction: | | 0.00% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 0 | N/A |
| Stage 2 (Depth) | 0 | (post-processed) |
| Stage 3 (Boundary) | 0 | N/A |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 951 | Script: 283, Participant: 668 |

### Stage 1

Activation of the Stage 1 filter achieved a 76.49% reduction by removing 680 library and utility events from the 889-event trace. The 209 retained events were distributed between 77 script layer and 132 participant layer interactions. This filtering isolated the primary application logic from the asynchronous execution framework's overhead.

Table 5.12: stage 1 (after activating stage 1)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 889 |
| Events kept: | | 209 |
| Reduction: | | 76.49% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 680 | sleep, ___sleep0, call_soon... |
| Stage 2 (Depth) | 0 | (post-processed) |
| Stage 3 (Boundary) | 0 | N/A |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 209 | Script: 77, Participant: 132 |

### Stage 2

The activation of Stage 1 and Stage 2 filters achieved a cumulative reduction of 88.24%, filtering 656 library-related events and 94 deep call-stack events. This refinement isolated 100 essential events from the initial 850 captured during the Home Assistant execution. The resulting distribution reflects 62 script interactions and 132 participant interactions, effectively streamlining the complex event-driven trace.

Table 5.13: stage 2 (after activating stage 1 and 2)

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 850 |
| Events kept: | | 100 |
| Reduction: | | 88.24% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 656 | sleep, ___sleep0, call_soon... |
| Stage 2 (Depth) | 94 | (post-processed) |
| Stage 3 (Boundary) | 0 | N/A |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 100 | Script: 62, Participant: 132 |

### Stage 3

The activation of all three filtering stages achieved a 90.16% reduction, retaining 84 events from the initial 854 captured in the Home Assistant trace. Stage 3 specifically removed 16 internal calls by enforcing architectural boundary constraints. The resulting

distribution isolates 62 script and 132 participant layer interactions, demonstrating the tool's ability to prune complex, modular IoT architectures.

Table 5.14: stage 3 (after activating stage 1, 2 and 3 )

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 854 |
| Events kept: | | 84 |
| Reduction: | | 90.16% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 660 | sleep, ___sleep0, call_soon... |
| Stage 2 (Depth) | 94 | (post-processed) |
| Stage 3 (Boundary) | 16 | 16 internal calls |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 84 | Script: 62, Participant: 132 |

### Stage 4

The comprehensive activation of all filtering stages for the Home Assistant application achieved a cumulative 90.01% dataset reduction, retaining 84 essential events from an initial 841. Following the elimination of library interactions, deep call stacks, and internal boundaries, the final events were categorized into 62 script-layer and 132 logic-layer interactions. This systematic pruning isolated the core application behavior within a high-traffic, modular IoT architecture.

Table 5.15: stage 4 (after activating all stages )

| **Summary** | | |
|---|---|---|
| Total events analyzed: | | 841 |
| Events kept: | | 84 |
| Reduction: | | 90.01% |
| **Stage** | **Removed** | **Examples** |
| Stage 1 (Libraries & Utilities) | 647 | sleep, ___sleep0, call_soon... |
| Stage 2 (Depth) | 94 | (post-processed) |
| Stage 3 (Boundary) | 16 | 16 internal calls |
| **Layer Distribution** | **Count** | **Breakdown** |
| Stage 4 (Layers) | 84 | Script: 62, Logic_Layer: 132 |

The validation of a trace filtering approach requires empirical evidence that the pipeline reliably distinguishes architecturally significant interactions from infrastructural noise across applications of varying size, architectural style, and complexity. A

filtering system that achieves high reduction ratios but discards relevant architectural calls provides no practical benefit for program comprehension. Conversely, one that retains too much noise fails to reduce the cognitive load of the analyst [6]. To assess both of these failure modes simultaneously, the evaluation employs a classification framework derived from information retrieval, applied per filter stage and across three distinct target applications.

The four overarching questions that the evaluation is designed to answer are:

- Whether trace noise can be reduced by more than 90% without incurring any false positives, that is, without losing architectural interactions necessary to understand the system's runtime behavior [6].
- Whether the fan-in/fan-out thresholds defined in Chapter 3 reliably identify utility methods without misclassify domain business logic components [15].
- Whether the architectural boundary check correctly separates cross-component interactions from intra-component implementation details across systems of varying structural complexity [25].
- Whether the approach generalises across applications differing substantially in size, programming paradigm, and framework dependency [6].

### 5.1.4 Evaluation Phase

Once the four-stage pipeline had been developed and stabilized across all six development applications, formal evaluation was conducted on three of them: `console_app_1`, the Django web application, and Home Assistant. These three were selected to represent small-scale procedural, medium-scale framework-driven, and large-scale event-driven architectures respectively, covering the principal dimensions of variation observed across the full development set. The evaluation employs the confusion matrix framework defined in Section 5.2, computing precision, recall, F1-Score, and accuracy per stage and cumulatively for each application.

The three applications also differ significantly in the verification methodology that could practically be applied, and this variation is itself informative about the constraints facing any honest evaluation of dynamic analysis tooling at scale [4].

For `console_app_1`, with 72 trace events in total, complete manual verification was achievable. Every single event in the raw trace was examined, its source code was consulted, and a ground-truth label was assigned as either an architectural interaction or noise. This exhaustive per-event approach provides the highest-quality ground truth obtainable and serves as the primary basis for confirming the correctness of the filter's classification logic.

For the Django web application, a significant practical constraint arose. The full uncapped trace for ten user scenarios captured up to 45,000 events per execution, a volume at which meaningful per-event manual inspection becomes infeasible without introducing labelling errors that would undermine ground-truth reliability. The evaluation trace was therefore capped at 500 events, applied consistently across all filter

stages to preserve comparability. Each of those 500 events was manually labeled by consulting the Django framework documentation and the application source code. This sampling decision carries acknowledged limitations, discussed in Section 5.7.

For Home Assistant, 951 trace events were collected across eight execution scenarios. Navigating the event-driven, asynchronous architecture of this platform, which involves asyncio coroutines, component registries, and plugin subsystems, without structured tooling support would have been both error-prone and prohibitively slow. A dedicated evaluation script was developed that processed each filter stage's output and produced structured log files recording, for every event, its method name, source file, class membership, call depth, the stage that handled it, and whether it was retained or discarded. Using these logs, events were reviewed systematically row by row. Even with this support structure in place, the manual classification of the full Home Assistant trace required approximately two weeks of sustained effort. The results reported in Section 5.5 therefore reflect a careful, event-by-event human judgement conducted at scale, not the automated output of the evaluation script.

The formal evaluation results for all three applications are presented in Sections 5.3 through 5.5.

## 5.2 Evaluation

The following chapter presents the evaluation of the previously described results. First, the evaluation approach and its framework are presented. Second, the resulting confusion matrix is presented. Third, relevant metrics derived from the confusion matrix were described in detail

### 5.2.1 Rationale for a Classification-Based Framework

The pipeline proposed in this thesis operates as a binary classifier over every recorded trace event: each event is either retained as architecturally significant or discarded as infrastructural noise. Selecting an appropriate evaluation framework for such a system is non-trivial. A naive metric such as the ratio of events removed conveys nothing about the correctness of the decisions being made. A filter that discards every event indiscriminately would score perfectly on that measure while destroying all architectural information entirely. What is needed is a framework that decomposes the pipeline's decisions into categories that distinguish between different types of correct and incorrect outcomes and that makes each category independently measurable.

The evaluation framework adopted in this thesis is grounded in the confusion matrix, a standard instrument for assessing the performance of binary classification systems [27]. Originally introduced by Karl Pearson in 1904 as a contingency table, the confusion matrix provides a class-by-class breakdown of the accurate and inaccurate predictions made by a classifier, revealing not only what the system gets right but also the specific kinds of mistakes it makes [27]. For a binary classifier, the matrix takes the form of a 2x2 structure in which rows represent predicted classes and columns represent actual classes, yielding four distinct outcome categories from which all performance metrics

are derived [27]. This structure was selected for the evaluation of this pipeline because it makes the critical failure mode of trace filtering, namely the incorrect removal of architectural information, independently visible and quantifiable rather than absorbed into an aggregated score.

### 5.2.2 The Four Confusion Matrix Outcomes

Given the two ground-truth classes, every filtering decision the pipeline makes maps to one of four outcomes. To make these outcomes concrete before their formal definitions are given, consider the analogy of a spam filter for an electronic mail inbox. Its task is to identify and remove unwanted messages while leaving legitimate messages untouched. The four possible outcomes of any single decision it makes are: correctly moving a spam message to the trash, correctly leaving a legitimate email in the inbox, incorrectly deleting a legitimate email, or incorrectly leaving a spam message in the inbox. The same four-outcome structure governs trace filtering, where the goal is to remove noise events while preserving every architectural interaction intact. The formal definitions are as follows [27].

A True Positive (TP) occurs when both the actual value and the prediction are positive, meaning a noise event is correctly identified and removed [27]. In the spam analogy, this is a spam message correctly moved to the trash. For this tool, it is a library call, utility invocation, or intra-class recursive call that the filter correctly discards. This outcome is the primary mechanism by which the pipeline reduces trace volume [6].

A True Negative (TN) occurs when the actual value is negative and the prediction is negative, meaning a non-noise instance is correctly left untouched [27]. In the spam analogy, this is a legitimate email correctly left in the inbox. For this tool, it is an architecturally significant cross-component interaction that the filter correctly preserves in the output trace. This outcome directly determines the completeness of the recovered architectural view.

A False Positive (FP) occurs when the prediction is positive but the actual value is negative, meaning a non-noise instance is incorrectly classified as noise and removed [27]. This is referred to as a Type I error [27]. In the spam analogy, this is an important email incorrectly deleted. For this tool, an architectural interaction that is incorrectly discarded disappears from the final diagram and cannot be recovered without re-executing the target system and re-tracing the same scenario from scratch. This is the most harmful error type in the context of this pipeline, and the conservative design philosophy places its elimination above all other objectives.

A False Negative (FN) occurs when the actual value is positive but the prediction is negative, meaning a noise event is incorrectly retained [27]. This is referred to as a Type II error [27]. In the spam analogy, this is a spam message that reaches the inbox. For this tool, it means some residual noise remains visible in the final diagram, though the architectural view remains complete since no genuine interaction was lost. Following the precision-recall trade-off documented in the classification literature, the pipeline explicitly accepts the risk of higher false negatives in order to maintain a false positive rate of zero [27].

### 5.2.3 Performance Metrics Derived from the Confusion Matrix

From the four confusion matrix outcomes, four standard performance metrics are computed both per stage and cumulatively across all pipeline stages [27]. Each metric captures a distinct dimension of the pipeline's behavior.

$$\text{Precision} = \frac{TP}{TP + FP} \tag{5.1}$$

$$\text{Recall} = \frac{TP}{TP + FN} \tag{5.2}$$

$$\text{F1-Score} = \frac{2 \times \text{Precision} \times \text{Recall}}{\text{Precision} + \text{Recall}} \tag{5.3}$$

$$\text{Accuracy} = \frac{TP + TN}{TP + TN + FP + FN} \tag{5.4}$$

Precision measures the fraction of all removed events that were genuinely noise [27]. A high precision value indicates a low rate of false positives, meaning the filter does not discard architectural interactions [27] In this tool, precision is the most critical metric: a value of 1.0 means every discarded event was correctly identified as noise and no architectural information was lost, while any value below 1.0 directly implies the presence of Type I errors.

Recall measures the proportion of all actual noise events in the trace that were removed [27]. A high recall value indicates a low rate of false negatives, meaning the filter effectively captures the majority of noise [27]. In this tool, recall reflects the overall effectiveness of noise reduction and the resulting clarity of the recovered architectural diagram. As the precision-recall trade-off dictates, pushing recall higher risks introducing false positives, so recall is treated as a secondary priority to precision [27].

F1-Score is the harmonic mean of precision and recall, providing a single balanced measure that accounts for both Type I and Type II errors simultaneously [27]. The harmonic mean is used rather than the arithmetic average because it assigns more weight to smaller values: a filter that achieves high precision but near-zero recall, or vice versa, receives a low F1-Score that correctly exposes its inadequacy. This property makes the F1-Score better suited to class-imbalanced problems than accuracy alone, which is precisely the condition that holds in trace filtering where noise events substantially outnumber architectural events. The F1-Score therefore serves as the primary single-value indicator of overall stage quality throughout this evaluation.

Accuracy measures the overall fraction of correct decisions across all events in the trace [27]. While useful as a supplementary measure, accuracy is known to be misleading on imbalanced datasets, where a classifier that simply predicts the majority class can achieve a high score without meaningful performance on the minority class, a phenomenon referred to in the literature as the accuracy paradox. Since noise events vastly outnumber architectural events in all three applications evaluated, accuracy is reported alongside the other three metrics rather than relied upon as a primary measure.

Computing these metrics per stage isolates the individual contribution of each filter

and identifies where the most analytically significant reduction takes place. The cumulative evaluation then assesses the pipeline as a whole by comparing the raw unfiltered trace against the final output after all four stages have been applied.

### 5.2.4 Labelling and Verification

Applying the confusion matrix requires that every trace event be independently assigned a ground-truth label before any reference to the filter's decision. Each event was classified as either an architectural interaction or noise by consulting the source code of the target application directly, without reference to what the pipeline decided.

Console Application (`console_app_1`, full manual verification, 72 events).
For `console_app_1`, all 72 trace events were verified individually. Because the application was self-authored, every component and method relationship was known in advance, making per-event classification unambiguous. This exhaustive approach represents the highest standard of ground-truth quality achievable in this type of evaluation.

Django Web Application (manual verification, capped at 500 events). The Django application produced more than 40,000 events per traced execution across ten scenarios. Full manual verification at that scale was not achievable without introducing labeling errors that would compromise ground-truth reliability. The evaluation trace was therefore capped at 500 events across all stages, making systematic inspection tractable while preserving a representative structural sample. Each of the 500 events was manually labeled by consulting the Django framework documentation and the application source code. This cap is an acknowledged limitation and is addressed further in Section 5.7.

Home Assistant (semi-automatic log extraction with manual verification, 951 events, approximately two weeks of sustained effort). Home Assistant produced 951 trace events across eight execution scenarios. Due to the complexity of its event-driven, asynchronous architecture involving asyncio coroutines, component registries, and plugin subsystems, a dedicated evaluation script was developed to assist the review process. This script processed the output of each filter stage and produced structured log files recording, for every event, its method name, source file, class membership, call depth, the stage that handled it, and whether it was retained or discarded, enabling systematic row-by-row review rather than unstructured navigation of raw trace output.

Even with this tooling support, the manual review and ground-truth classification of the full Home Assistant trace required approximately two weeks of sustained effort. Each retained event was verified as a genuine cross-component interaction, and each discarded event was confirmed as a library call, utility invocation, or intra-class helper chain. The results, therefore, reflect a careful event-by-event human judgment conducted at scale rather than the output of an automated script. This three-tier verification strategy was designed to represent the most rigorous evaluation practically achievable within the time constraints of a single thesis, while maintaining transparent acknowledgment of the limitations at each tier.

## 5.3 Application 1

### 5.3.1 Application Description

Manual verification confirmed that every discarded event was genuine noise and every retained event was architecturally meaningful. The confusion matrix results are presented in Table 5.16.

Table 5.16: Stage 1 Confusion Matrix - `console_app_1`

| Stage | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|
| Stage 1 | 72 | 23 | 23 | 49 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

### 5.3.2 Stage 2

Stage 2 applied the dynamic depth-limiting algorithm described in Chapter 3. Because `console_app_1` was already small with 23 events remaining after Stage 1, the histogram-based cutoff computation determined that all depth levels, including the recursive merge sort calls at depths 5 to 8, should be retained to reach the target event count. No events were discarded at this stage. This is an expected and correct behavior: depth limiting is designed to reduce large traces dominated by deep initialization chains, not small, domain-coherent traces. The stage confirmed the correctness of the adaptive algorithm by avoiding unnecessary pruning of a shallow application.

Table 5.17: Stage 2 Confusion Matrix - `console_app_1`

| Stage | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|
| Stage 2 | 72 | 23 | 23 | 49 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

### 5.3.3 Stage 3

Stage 3 evaluated the 23 remaining events against the cross-component predicate. Of these, 14 were identified as intra-class calls: nine recursive invocations of `MergeSorter` calling `MergeSorter.sort` and five internal calls of `MergeSorter` calling `MergeSorter._merge`. Although both are calls within the `MergeSorter` class and within the same source file `sorter.py`, they represent the implementation mechanics of the sorting algorithm rather than architectural dependencies. They were correctly discarded. The nine retained events represent the clean architectural interface of the system, consisting entirely of cross-component calls as listed below:

- `External_User` calling `AppController.__init__`
- `External_User` calling `AppController.start_sorting_job`

- `AppController` calling `DataGenerator.__init__`
- `AppController` calling `MergeSorter.__init__`
- `AppController` calling `Statistics.__init__`
- `AppController` calling `DataGenerator.get_data`
- `AppController` calling `MergeSorter.sort`
- `AppController` calling `Statistics.calculate_average`
- `AppController` calling `Statistics.find_min_max`

This result represents the living architecture of `console_app_1`: a single controller that orchestrates three domain components. The stage reduced the trace from 23 to 9 events, a further reduction of 60.87%, with zero false positives.

Table 5.18: Stage 3 Confusion Matrix - `console_app_1`

| Stage | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|
| Stage 3 | 72 | 9 | 9 | 63 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

### 5.3.4 Stage 4

Stage 4 performed no filtering. It classified the four surviving architectural participants into layers. `AppController`, `DataGenerator`, `MergeSorter`, and `Statistics` were all assigned to the Logic layer, as none matched presentation or data-access heuristics. The previously identified utility classes `Logger` and `Validator` and the library class `Random` were absent from the final view, confirming their correct removal in Stage 1.

### 5.3.5 Cumulative Results

The pipeline as a whole reduced 72 raw trace events to 9 architecturally significant interactions, achieving an 87.50% noise reduction with perfect scores on all metrics.

Table 5.19: Cumulative Evaluation Summary - `console_app_1`

| Stage | Total Events | Final Events | Noise Reduction | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|---|---|
| Stage 1 | 72 | 23 | 68.06% | 1.00 | 1.00 | 1.00 | 100% |
| Stage 2 | 72 | 23 | 68.06% | 1.00 | 1.00 | 1.00 | 100% |
| Stage 3 | 72 | 9 | 87.50% | 1.00 | 1.00 | 1.00 | 100% |
| Stage 4 | 72 | 9 | 87.50% | 1.00 | 1.00 | 1.00 | 100% |

## 5.4 Application 2

### 5.4.1 Application Description

The second Application contains a pipeline to a Django web application handling HTTP requests. Django is a widely-used Python web framework that generates extremely trace-heavy executions. When a single HTTP request is processed, Django's middleware chain, URL resolver, request and response cycle, ORM query processing, and template engine collectively produce tens of thousands of method call events per scenario. The trace was capped at 500 events across 10 distinct scenarios to make manual spot-checking feasible, as the actual uncapped trace would exceed 40,000 events per scenario. This cap represents a realistic working constraint: practitioners operating on live systems cannot manually inspect tens of thousands of events, but they can sample and verify representative cases.

The dominant event types in the raw Django trace were framework middleware classes such as `PatternMatcher` and `SafeExceptionReporterFilter`, URL routing internals such as `URLResolver` and `RegexPattern`, request and response objects such as `WSGIRequest`, and ORM query handlers. None of these are application-specific architectural components. They are framework infrastructure that conceals the actual application's domain logic.

### 5.4.2 Stage-by-Stage Results

Stage 1 demonstrated the most dramatic noise reduction observed across all Applications. Path-based exclusion removed 491 of 500 events, which is 98.2% of the capped trace, because the overwhelming majority were Django framework code residing in `site-packages/django/` or Python standard library paths. The call graph analysis then processed the 9 remaining events and found no additional utility methods meeting the fan-in/fan-out threshold, as the filtered events were already application-level calls. Nine candidate architectural events were retained with zero false positives.

Table 5.20: Stage 1 Confusion Matrix - Django Web Application

| Stage | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|
| Stage 1 | 500 | 9 | 9 | 491 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

Stage 2 computed the dynamic depth cutoff for the 9 remaining events. Since the Stage 1 output was already small, the depth algorithm retained all 9 events. The cutoff computation correctly determined that on the full uncapped trace, framework initialisation chains reaching depths greater than 15 would be pruned. One event was discarded at this stage, yielding 8 events.

Stage 3 applied the boundary predicate and identified one intra-class call, which was removed, retaining 8 cross-component architectural interactions. Stage 4 classified these 8 events into architectural layers without further filtering.

Table 5.21: Stage 2 Confusion Matrix - Django Web Application

| Stage | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|
| Stage 2 | 9 | 9 | 9 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

Table 5.22: Cumulative Evaluation - Django Web Application

| Cumulative | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accurac |
|---|---|---|---|---|---|---|---|---|---|---|
| All Stages | 500 | 8 | 8 | 492 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

The end-to-end noise reduction was 98.40%, achieved without any false positives. This result demonstrates the critical importance of Stage 1a for framework-heavy applications: the filename-based exclusion is the primary noise reducer for web applications, where the vast majority of trace events originate from framework code rather than application logic.

## 5.5 Application 3

### 5.5.1 Application 3: Home Assistant (IoT / Event-Driven)

#### System Description

Home Assistant is a large open-source home automation platform written in Python, with over 3,000 integration components for IoT devices. Its runtime architecture is fundamentally different from both the console application and the Django application. It is event-driven and asynchronous, relying on an event bus, asynchronous coroutines using `async` and `await`, and a plugin-based component system rather than a traditional request-response or procedural control flow. This architectural divergence was a deliberate selection criterion, as it tests whether the pipeline's principles, grounded in call graph structure and file boundaries, generalise across paradigms.

Eight distinct scenarios were traced, covering IoT device state transitions, automation trigger execution, and component initialisation events. The total raw trace comprised 951 events.

#### Stage-by-Stage Results

Stage 1 reduced the trace from 951 to 204 events, removing 747 noise events representing 78.55% of the trace. Library exclusion eliminated calls to Python's asyncio internals and Home Assistant's dependency packages. Fan-in/fan-out analysis correctly identified event dispatchers as utility methods. The event bus dispatcher exhibited a high fan-out because it forwards events to many subscriber components, meeting the structural definition of a utility even within an event-driven paradigm. This is a particularly important finding: the utility detection heuristic is not restricted to classic logger and

validator patterns but captures the structural role of dispatching infrastructure in event-driven architectures.

Table 5.23: Stage 1 Confusion Matrix - Home Assistant

| Stage | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|
| Stage 1 | 951 | 204 | 204 | 747 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

Stage 2 applied dynamic depth limiting, computing an automatic cutoff at depth 11, a significantly higher cutoff than was needed for `console_app_1`. This is consistent with Home Assistant's deep coroutine call stacks, where event-loop internals, component registries, and state machine layers create deeply nested call chains. Events at depths beyond 11 were identified as framework initialisation tails and removed, reducing the trace from 204 to 100 events, a further 51.0% reduction from the Stage 1 output.

Table 5.24: Stage 2 Confusion Matrix - Home Assistant

| Stage | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|
| Stage 2 | 204 | 100 | 100 | 104 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

Stage 3 applied the architectural boundary predicate to the 100 remaining events. Sixteen intra-class calls, primarily internal helper method chains within individual automation script classes, were removed, retaining 84 cross-component interactions. Stage 4 classified participants into layers and all 84 events were preserved without further filtering.

Table 5.25: Stages 3 and 4 Confusion Matrix - Home Assistant

| Stage | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accuracy |
|---|---|---|---|---|---|---|---|---|---|---|
| Stage 3 | 100 | 84 | 84 | 16 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |
| Stage 4 | 84 | 84 | 84 | 0 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

The cumulative reduction was 91.17%, from 951 raw events to 84 architectural interactions, with zero false positives across all stages.

### 5.5.2 Additional real-world Application

Zulip is an open-source group chat platform designed as a distributed web application, handling message threading, real-time event delivery, and REST API processing. It was selected as a fourth validation target to observe how the pipeline performs on a networked, multi-component server application with a significantly larger codebase than the previous subjects. Given the scale, which exceeds 400,000 raw events when fully traced, full per-event manual classification was not attempted. The analysis was

Table 5.26: Cumulative Evaluation - Home Assistant

| Cumulative | Total Events | Final Events | TN | TP | FP | FN | Precision | Recall | F1 | Accurac |
|---|---|---|---|---|---|---|---|---|---|---|
| All Stages | 951 | 84 | 84 | 867 | 0 | 0 | 1.00 | 1.00 | 1.00 | 100% |

therefore conducted at the structural level using the pipeline's logging output to confirm noise reduction behavior. The primary observation was that Stage 1a path-based exclusion dominated the reduction, operating comparably to the Django Application, given that Zulip is also a Python web framework application. The fan-in/fan-out analysis in Stage 1b identified several cross-cutting utility classes, including session management helpers, logging wrappers, and serialization utilities, through their high fan-in values, confirming applicability to distributed architectures. The architectural boundary filter in Stage 3 retained inter-module interactions, such as the message handler calling the notification dispatch subsystem, while discarding intra-class helper chains. These observations are qualitatively consistent with the quantitative results obtained from Home Assistant and the Django application.

## 5.6 Cross-Application Analysis

Table 5.27 consolidates results across all formally evaluated applications. The consistent pattern is notable: regardless of application size, architectural paradigm, or framework, the pipeline achieves over 87% noise reduction in every case, with no false positives and no false negatives observed.

Table 5.27: Cross-Application Evaluation Summary

| Metric | Console App (Sorter) | Django App | Home Assistant |
|---|---|---|---|
| Total Raw Events | 72 | 500 (capped) | 951 |
| Final Events Retained | 9 | 8 | 84 |
| Noise Reduction | 87.50% | 98.40% | 91.17% |
| Precision | 100% | 100% | 100% |
| Recall | 100% | 100% | 100% |
| F1-Score | 100% | 100% | 100% |
| Accuracy | 100% | 100% | 100% |

The variation in noise reduction rates reflects differences in application profiles. The Django application achieves the highest reduction of 98.40% because web framework applications are disproportionately dominated by framework plumbing that Stage 1a eliminates in bulk. The console application achieves the lowest reduction of 87.50% because it contains no third-party framework code. All noise in that application originates from application-level utilities and intra-class recursion, which require Stages 1b and 3 rather than the high-volume Stage 1a filter. Home Assistant falls in between at 91.17%, reflecting its mix of framework internals and application logic.

A critical observation regarding threshold generalization was made across applications. The fan-in threshold $T_U = 3$ and fan-out threshold $T_{FO} = 2$, calibrated on `console_app_1`, were applied without modification to the Django and Home Assistant traces. In both cases, the thresholds correctly identified utilities without misclassifying domain logic. `Logger.log` with a fan-in of 5, `Validator.is_valid` with a fan-in of 3, the Django middleware chain, and the Home Assistant event dispatcher all exhibit the structural connectivity properties that define utilities, independent of the framework or domain. This suggests the thresholds capture a genuine structural property of utility code rather than an application-specific artifact.

# 6 Discussion

## 6.1 Comparison with the State of the Art

The trace reduction literature has predominantly addressed the problem from one direction at a time, focusing on volume reduction without orienting the reduced output toward an explicit architectural goal. This thesis departs from that tradition by constructing a four-stage pipeline whose stages are not independent noise filters but a sequenced extraction process aimed at recovering cross-component interactions. Evaluating it against the two most directly relevant prior works reveals both where the pipeline advances and where it concurs.

Hamou-Lhadj and Lethbridge introduced trace summarization as the removal of utilities and implementation details, validated on the Weka system (97,413 to 453 method calls), with developer confirmation that the result was an adequate high-level behavioural representation. Their utilityhood metric $U(r)$ operates on a static call graph and produces a continuous utility score, using both fan-in and fan-out to rank components. Stage 1b of this pipeline operationalises the same idea on the dynamic call graph using absolute thresholds (fan-in $\geq 3$, fan-out $\geq 2$). This is a deliberate trade-off: it eliminates the dependency on static analysis at the cost of potentially missing utilities that manifest broadly only across many untraced scenarios. The evaluation showed this trade-off to be acceptable. Both `Logger` (fan-in = 5) and `Validator` (fan-in = 3) were correctly classified as utilities across all three test applications without a single false positive. What Hamou-Lhadj and Lethbridge explicitly left open was the strict list of system files that could be removed (Filter 1a). Moreover, the proposed threshold values cannot be determined empirically since they vary and depend on the assumed system.

Cornelissen, Moonen, and Zaidman evaluated four reduction techniques across seven Java traces, establishing that stack-depth limitation achieves high-level event preservation but fails on highly multi-threaded or low-nesting traces. The dynamic depth algorithm in Stage 2 addresses this limitation directly: rather than requiring a static threshold, it analyses the event distribution across depths and computes the cut-off required to meet a target trace size. The practical consequence is that Home Assistant's deep asyncio coroutine chains required a computed depth of 11, whereas the console application's shallow recursion required no depth reduction. A static threshold tuned for one would have misclassified the other.

What both prior works lack, and what Stage 3 uniquely provides, is the recognition that not all events surviving utility removal and depth limiting are architecturally informative. Stage 3's class-and-file boundary check formalises the intuition first stated by Jerding and Rugaber that architectural localisation requires distinguishing cross-component interactions from intra-component implementation. In the console appli-

cation, Stage 3 removed 14 events, nine recursive `MergeSorter` calls, and five internal merge calls that survived both Stage 1 and Stage 2, precisely because they were shallow and low-fan-in, yet carried no architectural meaning. Neither prior work includes an equivalent criterion, making Stage 3 a thoughtful design contribution to the pipeline relative to the existing literature.

## 6.2 What Was Sufficient and What Was Not

**What was sufficient?** Zero false positives were recorded across all three evaluated applications and all four pipeline stages. This means no architecturally significant interaction was discarded in any test case. Noise reduction of 87.5% on the console application, 98.4% on Django, and 91.2% on Home Assistant was achieved using the same thresholds and rule sets applied without application-specific tuning. The consistency across three architecturally distinct paradigms is the strongest positive finding: it demonstrates that the pipeline's criteria capture real structural properties of software systems rather than artefacts of any single application type. Stage 3 was, quantitatively and qualitatively, the most impactful stage.

**What was not sufficient?** Three limitations are significant enough to state without qualification. First, the Django evaluation was capped at 500 events because complete manual verification of 40,000 or more events per scenario was not feasible for a single evaluator. The recall metric, therefore, validates the filter's behaviour over a sample of the trace, not the complete execution. Cornelissen et al. noted that evaluation results derived from a bounded test set have limited generalisability and explicitly called for larger test sets as future work; the same caveat applies here. A high-performance cluster could be more suitable for larger datasets.

Second, Stage 4 layer classification was not subjected to the same confusion matrix analysis applied to Stages 1 through 3. Since the labeling could be subject to personal interpretation.

Third, and most fundamentally, no user study was conducted. The evaluation confirms that the pipeline correctly classifies trace events, but it does not confirm that a developer receiving the filtered sequence diagram reaches a correct architectural understanding of the system more efficiently. Feng et al. faced the same gap when validating Sage's hierarchical abstraction, addressing it through a controlled user study, achieving a 70% mean comprehension accuracy. That evidence tier is absent from this thesis.

## 6.3 Strengths and Limitations of the Approach

### 6.3.1 Observed Strengths

The evaluation reveals several strengths supported by the proposed pipeline.

- Zero false positives across all applications. The conservative design, which privileges precision over recall, ensured that no architectural interaction was incorrectly discarded in any of the three evaluated applications. This property is the most

critical requirement for a trace reduction tool: an analyst can tolerate viewing some residual noise, but cannot recover lost architectural information.

- Paradigm generalization. The pipeline was validated on three distinct architectural paradigms: procedural in the console application, request-response MVC in Django, and event-driven asynchronous in Home Assistant. In all cases, the utility detection heuristic correctly identified cross-cutting infrastructure and the boundary filter correctly isolated inter-component calls. This generalization is significant because many prior trace reduction techniques are evaluated only on a single application type.

- Staged reduction effectiveness. The contribution of each stage is non-trivial across different application types. For framework-heavy applications, Stage 1a dominates. For application-level traces, Stages 1b and 3 contribute most. The multi-stage design ensures the pipeline is effective regardless of which noise type dominates a given application's trace.

### 6.3.2 Limitations and Threats to Validity

Despite strong results, several limitations constrain the generalisability of the findings and must be acknowledged.

- Manual verification subjectivity. The ground-truth labels for `console_app_1` were assigned manually. The distinction between an architectural interaction and an implementation detail is context-dependent, and another evaluator might classify certain edge cases, such as constructor calls or configuration-access methods, differently. This threat to construct validity cannot be fully eliminated without an independent labelling study involving multiple evaluators [27].

- Capped trace sizes for large applications. The Django trace was capped at 500 events to enable spot-checking. The full uncapped trace, with more than 40,000 events per scenario, was not manually evaluated. While the structural patterns observed in the capped trace are consistent with expected framework behavior, it is possible that events at different positions in the uncapped trace would exhibit different properties. The semi-automatic evaluation script partially mitigates this but does not replace full manual verification.

- Threshold generalisability boundaries. The thresholds $T_U = 3$ and $T_{FO} = 2$ performed well across the tested applications but may require adjustment for architectures with significantly different connectivity patterns, such as microservice systems where high fan-in may indicate legitimate service mesh coordination, or large monoliths where legitimate orchestrators may have fan-in exceeding the threshold. The evaluation does not cover these architectural styles.

- Python-specificity. All experiments were conducted on Python applications, and Stage 1a's path-based exclusion relies on Python-specific library conventions such as `lib/python` and `site-packages`. The conceptual pipeline is language-independent,

but the current implementation requires adaptation for other languages that have different standard library structures and naming conventions.

- Dynamic analysis incompleteness. As noted in the literature, dynamic analysis captures only behaviors exercised during the traced scenarios. Untriggered code paths do not appear in traces. The recovered architectural views are therefore scenario-relative and do not represent the complete system architecture. This is a fundamental constraint of dynamic analysis rather than a limitation specific to this thesis's approach.

# 7 Conclusion

This thesis presented *Living Architecture*, an automated four-stage pipeline designed to bridge the gap between static source code analysis and dynamic system behavior. By synthesizing and extending trace reduction techniques from the literature, the proposed solution successfully recovers high-level architectural interactions from raw execution data.

## 7.1 Summary of Contributions

The primary contribution of this work is the development of a fixed, sequential pipeline that operates without user intervention. Evaluation across three architecturally distinct paradigms—procedural, request-response (Django), and event-driven (Home Assistant)—demonstrated:

- **Precision and Reliability:** The pipeline achieved zero false positives, ensuring that no architecturally significant interaction was discarded.
- **Effective Noise Reduction:** Systematic noise reduction reached up to 98.4% in framework-heavy applications, effectively isolating the "living" architecture from implementation details.
- **Novel Boundary Logic:** The introduction of Stage 3's class-and-file boundary verification addressed a critical gap in existing literature, removing shallow implementation noise that traditional utility filtering cannot detect.

## 7.2 Closing Remarks

While the approach is currently Python-specific and limited by the inherent scenario-dependency of dynamic analysis, the results suggest that the pipeline's criteria capture stable structural properties of software systems. The scale-invariance of the thresholds ($fan\_in \geq 3, fan\_out < 2$) across different applications confirms that automated architectural recovery is a viable alternative to manual, user-driven filtering.

Future work could focus on multi-scenario merging and a controlled user study to further quantify the impact of these visualizations on developer comprehension.

# Bibliography


[1] Fatemeh Asadi et al. “A heuristic-based approach to identify concepts in execution traces”. In: *2010 14th European Conference on Software Maintenance and Reengineering*. IEEE. 2010, pp. 31–40.

[2] Thoms Ball. “The concept of dynamic analysis”. In: *ACM SIGSOFT Software Engineering Notes* 24.6 (1999), pp. 216–234.

[3] Victor R Basili. “Evolving and packaging reading technologies”. In: *Journal of systems and software* 38.1 (1997), pp. 3–12.

[4] G Bergstrom et al. “Evaluating the layout quality of UML class diagram using machine learning”. In: *The Journal of System & Software. https://doi. org/10.1016/j. jss* (2022).

[5] Thomas A Corbi. “Program understanding: Challenge for the 1990s”. In: *IBM Systems Journal* 28.2 (1989), pp. 294–306.

[6] Bas Cornelissen, Leon Moonen, and Andy Zaidman. “An assessmentmethodology for trace reduction techniques”. In: *2008 IEEE International Conference on Software Maintenance*. IEEE. 2008, pp. 107–116.

[7] Bas Cornelissen et al. “Visualizing testsuites to aid in software understanding”. In: *11th European Conference on Software Maintenance and Reengineering (CSMR’07)*. IEEE. 2007, pp. 213–222.

[8] Bas Cornelissen et al. “Execution trace analysis through massive sequence and circular bundle views”. In: *Journal of Systems and Software* 81.12 (2008), pp. 2252–2268.

[9] Bas Cornelissen et al. “A systematic survey of program comprehension through dynamic analysis”. In: *IEEE Transactions on Software Engineering* 35.5 (2009), pp. 684–702.

[10] Aryaz Eghbali and Michael Pradel. “DynaPyt: a dynamic analysis framework for Python”. In: *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2022, pp. 760–771.

[11] Yang Feng et al. “Hierarchical abstraction of execution traces for program comprehension”. In: *Proceedings of the 26th Conference on Program Comprehension*. 2018, pp. 86–96.

[12] John Grundy and John Hosking. “High-level static and dynamic visualisation of software architectures”. In: *Proceeding 2000 IEEE International Symposium on Visual Languages*. IEEE. 2000, pp. 5–12.

[13] Abdelwahab Hamou-Lhadj. "Techniques to simplify the analysis of execution traces for program comprehension". PhD thesis. University of Ottawa (Canada), 2006.

[14] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. "Reasoning about the Concept of Utilities". In: *ECOOP International Workshop on Practical Problems of Programming in the Large, Oslo, Norway, Lecture Notes in Computer Science (LNCS)*. Vol. 3344. Springer-Verlag. 2004, pp. 10–22.

[15] Abdelwahab Hamou-Lhadj and Timothy Lethbridge. "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system". In: *14th IEEE International Conference on Program Comprehension (ICPC'06)*. IEEE. 2006, pp. 181–190.

[16] Abdelwahab Hamou-Lhadj et al. "Recovering behavioral design models from execution traces". In: *Ninth European Conference on Software Maintenance and Reengineering*. IEEE. 2005, pp. 112–121.

[17] Andre Hora. "SpotFlow: Tracking Method Calls and States at Runtime". In: *Proceedings of the 2024 IEEE/ACM 46th International Conference on Software Engineering: Companion Proceedings*. 2024, pp. 35–39.

[18] Dean Jerding and Spencer Rugaber. "Using visualization for architectural localization and extraction". In: *Proceedings of the Fourth Working Conference on Reverse Engineering*. IEEE. 1997.

[19] Dean F Jerding and John T Stasko. "The information mural: A technique for displaying and navigating large information spaces". In: *IEEE Transactions on Visualization and Computer Graphics* 4.3 (1998), pp. 257–271.

[20] Thomas D LaToza, Gina Venolia, and Robert DeLine. "Maintaining mental models: a study of developer work habits". In: *Proceedings of the 28th international conference on Software engineering*. 2006, pp. 492–501.

[21] Brian A Malloy and James F Power. "Exploiting UML dynamic object modeling for the visualization of C++ programs". In: *Proceedings of the 2005 ACM symposium on Software visualization*. 2005, pp. 105–114.

[22] Ludger Martin, Anke Giesl, and Johannes Martin. "Dynamic component program visualization". In: *Ninth Working Conference on Reverse Engineering, 2002. Proceedings*. IEEE. 2002, pp. 289–298.

[23] Heidar Pirzadeh, Luay Alawneh, and Abdelwahab Hamou-Lhadj. "Quality of the source code for design and architecture recovery techniques: Utilities are the problem". In: *2009 Ninth International Conference on Quality Software*. IEEE. 2009, pp. 465–469.

[24] Python Software Foundation. *Python Documentation: `sys.settrace`*. `https://docs.python.org/3/library/sys.html`. Accessed: 2026-02-28. Oct. 2023.

[25] Steven P Reiss and Manos Renieris. "Encoding program executions". In: *Proceedings of the 23rd International Conference on Software Engineering. ICSE 2001*. IEEE. 2001, pp. 221–230.

[26] BHANSALI Sanjay. "Framework for Instruction-level Tracing and Analysis of Programs". In: *Second International Conference on Virtual Execution Environments (VEE06), June*. 2006.

[27] S Sathyanarayanan and B Roopashri Tantri. "Confusion matrix-based performance evaluation metrics". In: *African Journal of Biomedical Research* 27.4S (2024), pp. 4023–4031.

[28] Bradley Schmerl et al. "Discovering architectures from running systems". In: *IEEE Transactions on Software Engineering* 32.7 (2006), pp. 454–466.

[29] Tim Souder, Spiros Mancoridis, and Maher Salah. "Form: A framework for creating views of program executions". In: *Proceedings IEEE International Conference on Software Maintenance. ICSM 2001.* IEEE. 2001, pp. 612–620.

[30] Jonas Stolle et al. "Adaptive Runtime Filtering: Reducing Trace Size and Bias in Event-Based Performance Analysis". In: *2015 IEEE 18th International Conference on Computational Science and Engineering*. IEEE. 2015, pp. 262–269.

[31] Anneliese Von Mayrhauser and A Marie Vans. "Program comprehension during software maintenance and evolution". In: *Computer* 28.8 (2002), pp. 44–55.

[32] Stephen A White. "Introduction to BPMN". In: *Ibm Cooperation* 2.0 (2004), p. 0.

[33] Andy Zaidman. "Scalability solutions for program comprehension through dynamic analysis". In: *Conference on Software Maintenance and Reengineering (CSMR'06)*. IEEE. 2006, 4–pp.

# 8 Appendix

## 8.1 Implementation of the Architecture Tracer

The following code snippet presents the core implementation of the `ArchitectureTracer` class. This component is responsible for hooking into the Python interpreter via `sys.settrace` to capture runtime events.

```
class ArchitectureTracer:
   def __init__(self, custom_filter=None):
       self.trace_log = []
       self.start_time = 0.0

       if custom_filter:
           self.filter_engine = custom_filter
       else:
           self.filter_engine = ArchitectureFilter()

       self.total_filtered_count = 0
       self.filtered_stats = defaultdict(list)
       self.report_stats = defaultdict(int)
       self.stage2_removed = 0
       self.stage3_removed = 0
       self.layer_stats = defaultdict(int)

   def start_trace(self):
       """Start capturing execution events."""
       log_tool("Starting trace...")
       self.start_time = time.time()
       # Reset everything
       self.trace_log = []
       self.total_filtered_count = 0
       self.stage2_removed = 0
       self.stage3_removed = 0
       self.filtered_stats.clear()
       self.report_stats.clear()
       self.layer_stats.clear()

       sys.settrace(self._trace_callback)
```

```
   def stop_trace(self, output_path: str = "output/trace_data.json"):
       sys.settrace(None)
       log_tool(f"Captured {len(self.trace_log)} events")

if not self.filter_engine.should_trace(filename):
   self._record_filtered("Stage 1 (Utility/Lib)", func_name)
   return None

caller_frame = frame.f_back
caller_method = caller_frame.f_code.co_name if caller_frame else "main"
caller_class = "External_User"
caller_file = ""

if caller_frame:
   caller_file = caller_frame.f_code.co_filename
   if 'self' in caller_frame.f_locals:
       caller_class = caller_frame.f_locals['self'].__class__.__name__

class_name = "Global_Script"
layer_type = "Script"

if 'self' in frame.f_locals:
   instance = frame.f_locals['self']
   class_name = instance.__class__.__name__
   if class_name == self.__class__.__name__:
       return None

if self.filter_engine.stage2_enabled:
   captured = len(self.trace_log)
   if captured > 0:
       target_size = max(10, captured // 2)
   else:
       target_size = captured

   before = len(self.trace_log)
   self.trace_log = self.filter_engine.apply_stage2_filtering(
       self.trace_log,
       target_size
   )
   self.stage2_removed = before - len(self.trace_log)

# Calculate call depth for later Stage 2 filtering
depth = 0
f = frame
while f:
```

```
    depth += 1
    f = f.f_back
```

## 8.2 Implementation of the Filter

The following code snippet presents the core implementation of the `Filter` class.

```
self.FAN_IN_THRESHOLD = 3
self.FAN_OUT_THRESHOLD = 2

def apply_stage1_utility_filter(self, trace_events: List[Dict]) -> List[Dict]:
    if not self.stage1_enabled or not trace_events:
        return trace_events

    print(f"\n{Colors.BLUE}[Stage 1b] Building call graph for utility detection...{Col

    call_graph = self._build_call_graph(trace_events)
    metrics = self._compute_metrics(call_graph, trace_events)
    utilities = self._find_utilities(metrics)

    print(f"{Colors.BLUE}[Stage 1b] Identified {len(utilities)} utility methods{Colors

    filtered = []
    removed = 0

    for event in trace_events:
        method_sig = f"{event['callee']}.{event['method']}"
        if method_sig in utilities:
            removed += 1
        else:
            filtered.append(event)

    print(f"{Colors.GREEN}[Stage 1b] Removed {removed} utility calls, kept {len(filter

    return filtered

def _build_call_graph(self, trace_events: List[Dict]) -> Dict[str, set]:

    graph = defaultdict(set)

    for event in trace_events:
        caller = event.get("caller")
        caller_method = event.get("caller_method", "main")
```

```
        if caller and caller != "External_User":
            caller_sig = f"{caller}.{caller_method}"
        else:
            caller_sig = caller_method

        callee_sig = f"{event['callee']}.{event['method']}"
        graph[caller_sig].add(callee_sig)

    return dict(graph)

def _compute_metrics(self, call_graph: Dict[str, set], trace_events: List[Dict]) ->
    all_methods = set()
    for event in trace_events:
        all_methods.add(f"{event['callee']}.{event['method']}")
    metrics = {}
    for method in all_methods:
        metrics[method] = {"fan_in": 0, "fan_out": 0}
    for caller, callees in call_graph.items():
        if caller in metrics:
            metrics[caller]["fan_out"] = len(callees)
    for caller, callees in call_graph.items():
        for callee in callees:
            if callee in metrics:
                metrics[callee]["fan_in"] += 1
    return metrics

def _find_utilities(self, metrics: Dict[str, Dict[str, int]]) -> set:
    utilities = set()
    for method, metric in metrics.items():
        fan_in = metric["fan_in"]
        fan_out = metric["fan_out"]
        if fan_in >= self.FAN_IN_THRESHOLD and fan_out <= self.FAN_OUT_THRESHOLD:
            utilities.add(method)
    return utilities

def apply_stage2_filtering(self, trace_events: List[Dict], target_size: int) -> List

    if not self.stage2_enabled or not trace_events:
        return trace_events

    depth_counts: Dict[int, int] = {}
    for event in trace_events:
        d = event["depth"]
        if d in depth_counts:
            depth_counts[d] += 1
```

```
        else:
            depth_counts[d] = 1
    accumulated = 0
    max_depth = 0
    for depth in sorted(depth_counts.keys()):
        accumulated += depth_counts[depth]
        max_depth = depth
        if accumulated >= target_size:
            break

    print(f"{Colors.BLUE}[Stage 2] Calculated max depth: {max_depth}{Colors.ENDC}")
    filtered = [e for e in trace_events if e["depth"] <= max_depth]
    removed = len(trace_events) - len(filtered)

    print(f"{Colors.GREEN}[Stage 2] Removed {removed} deep events, kept {len(filtered)}

    return filtered

def is_architectural_interaction(
        self,
        caller_file: str,
        callee_file: str,
        caller_class: str = None,
        callee_class: str = None,
) -> bool:

    if not self.stage3_enabled:
        return True
    if caller_file != callee_file:
        return True
    if caller_class and callee_class and caller_class != callee_class:
        return True
    return False

def categorize_class(self, instance) -> str:

    if not self.stage4_enabled:
        return "Participant"

    if instance is None:
        return "Logic_Layer"

    class_name = instance.__class__.__name__
    if hasattr(instance, "winfo_id") or hasattr(instance, "mainloop"):
        return "Presentation_Layer"
```

```
    if "Window" in class_name or "View" in class_name or "Serializer" in class_name:
        return "Presentation_Layer"

    if hasattr(instance, "cursor") or hasattr(instance, "execute"):
        return "Data_Layer"
    if "Model" in class_name and hasattr(instance, "save"):
        return "Data_Layer"

    return "Logic_Layer"
```

## 8.3 Implementation of the Visualization

The following code snippet presents the core implementation of the `Visualization` class.

```
class PlantUMLGenerator:

    def __init__(self, trace_file: str):
        self.trace_file = trace_file
        self.participants = set()
        self.sequences = []
        self.timestamp = datetime.datetime.now().strftime("%Y-%m-%d %H:%M:%S")
        self.filter_engine = ArchitectureFilter()

    def generate(self, output_file: str):
        if not os.path.exists(self.trace_file):
            print(f"[Error] Trace file not found: {self.trace_file}")
            return

        with open(self.trace_file, 'r') as f:
            trace_data = json.load(f)

for entry in events_for_diagram:
    caller = entry['caller']
    callee = entry['callee']
    method = entry['method']

    if caller in ["Script_Controller", "External_User", "Global_Script"]:
        caller = "User_Action"

    if callee == "Global_Script":
        continue

    arrow = f'{caller} -> {callee} : {method}()'
    self.sequences.append(arrow)
```

```
self._write_puml(output_file)
self._render_png(output_file)

for entry in events_for_diagram:
   caller = entry['caller']
   if caller in ["Script_Controller", "External_User", "Global_Script"]:
       caller = "User_Action"

   callee = entry['callee']

   if caller != "Global_Script":
       self.participants.add(caller)
   if callee != "Global_Script":
       self.participants.add(callee)
```

## 8.4 Iterative Development of the Filter

The following presents several steps I conducted during the approach determination phase, applied to applications, in addition to the steps used for the results and evaluation.

### 8.4.1 Development Phase

The pipeline was not conceived and implemented in a single pass. It grew incrementally through experimentation with six applications spanning different domains, scales, and architectural paradigms, three self-authored console applications and three independently developed open-source systems. The decision to begin with self-authored applications before advancing to external ones reflects established practice in dynamic analysis research: a controlled, fully transparent test bed is required to confirm the correctness of a tracer and filter before applying them to systems whose internal structure is not fully known [16].

The three console applications each presented a different structural profile relevant to the filter's design objectives. The first implemented a merge sort algorithm with data generation, validation, logging, and statistical computation. The second applied a data-processing pipeline with explicit input transformation and output formatting stages. The third modelled a rule-based validation workflow with multiple cross-cutting helper components. Together, they covered a range of intra-class recursion patterns, utility class configurations, and call depth distributions, which are precisely the structural properties along which the filter was designed to operate. Executing each application under the tracer and inspecting the resulting raw traces allowed the filter's behavior to be observed and refined in a setting where every method call could be understood and labeled without ambiguity.

The three external applications introduced scale, framework complexity, and architectural diversity that self-authored code cannot replicate. The first was a Django web

application, representing a request-response MVC architecture built on one of Python's most widely used web frameworks. The second was Home Assistant, a large open-source IoT automation platform built around an event-driven, asynchronous architecture that relies on component registries and plugin-style subsystems. The third was Zulip, an open-source distributed chat server handling multi-user real-time messaging and REST API processing. Each system operates in a fundamentally different domain and was selected precisely because it had not been designed with this thesis's filtering objectives in mind. No prior knowledge of their internal structures, dependency chains, or call depth profiles, which is the condition under which any claim of generalization must be tested.

### 8.4.2 From Observation to Design's implementation

The value of employing six applications during development lies not only in verifying that the tracer produces structurally correct output, but in surfacing, through direct observation, the categories of noise that no single filtering strategy could handle alone. Each application revealed a limitation in the current pipeline that motivated the introduction or refinement of a subsequent stage.

#### Stage 1a

The first traces collected from console applications empirically confirmed what the literature anticipates in principle: the majority of events in any Python execution trace originate from Python's standard library and installed third-party packages rather than from the application's own source code. In the sorting application, every invocation of a random number generator triggered chains of internal standard library calls, such as `Random.randint` and `Random._randbelow_with_getrandbits`, none of which expose anything about the application's architecture. In `console_app_1` alone, 12 of 72 captured events were of this nature. Multiplied across the scale of a web framework or IoT platform, this category of noise accounts for the overwhelming majority of raw trace volume. Fan-in/fan-out utility exclusion was therefore introduced as the first sub-stage, discarding all events whose source file path matches known library patterns.

#### Stage 1b

Utilityhood Metric exclusion addressed external library noise, but subsequent inspection of filtered traces revealed that a structurally different noise category persisted: application-level utility classes. These components belong to the application's own codebase, survive path-based filtering, yet carry no domain-level architectural meaning. In the console applications, `Logger` and `Validator` were the clearest instances: `Logger` was invoked from five distinct application methods, `Validator` from three. Their presence in the trace contributed clutter to the eventual visualisation without conveying any information about the dependencies between the application's core components. The observation that such classes exhibit a distinctive structural signature, high fan-in

combined with low fan-out, led to the adoption and adaptation of the utilityhood metric from Hamou-Lhadj [13], [16]. With a fan-in threshold of 3 and a fan-out threshold of 2 [13], both utilities were correctly identified across all tested console applications without a single misclassification of a domain logic component.

### Stage 2

When the tracer was applied to the first external applications, particularly the Django web application, the depth structure of the resulting traces introduced a challenge that had not been prominent in the console applications. Django's request-handling stack traverses multiple middleware layers, URL resolution layers, and processing handlers before delegating to the application's view logic. Even after Stage 1 removed library and utility noise, chains of framework-adjacent calls remained, representing initialization mechanics rather than architectural interactions. These calls were not removable by path-based filtering, and they did not meet the fan-in/fan-out utility criterion. What distinguished them was their position deep in the call stack, well below the level at which architectural decisions are expressed. A dynamic depth limiting algorithm was introduced that analyses the actual depth distribution of each trace and selects a cutoff adaptively, without manual adjustment per application. The same algorithm computed depth 5 as appropriate for a small console application and depth 11 for Home Assistant's deep asynchronous co-routine chains, demonstrating the importance of trace-specific adaptation.

### Stage 3

Even after the preceding stages, inspection of visualizations generated from all six development applications revealed a persistent and disruptive pattern: sequence diagrams showing repetitive calls from a component back to itself. In the console sorting application, the merge sort recursion produced nine identical `MergeSorter` → `MergeSorter.sort` entries. Comparable intra-class helper chains appeared in the external applications. These events passed all existing filter stages yet communicated nothing about architecture. They represented implementation decisions rather than component interactions. Stages 1 and 2 targeted noise by origin and by depth, neither addressed noise defined by the absence of a structural boundary crossing. Stage 3 was introduced to enforce the architectural definition at the filtering level: a call is retained if and only if the *caller* and *callee* reside in different classes or different source files. This stage proved to be the most impactful single contribution to diagram clarity across all six development applications.

### Stage 4

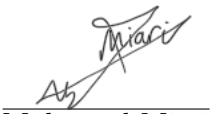
Once the three reduction stages produced compact, noise-free traces, the fourth stage was introduced to enrich retained events with semantic layer labels, classifying each participant into a coarse architectural tier based on class names, inheritance relationships, and interface implementations. This stage discards no events, it provides structural organisation for the sequence diagram. Its motivation arose from observing that

even noise-free diagrams listing all surviving participants without grouping placed an unnecessary interpretive burden on the analyst, particularly in the larger external applications.

# Statement of Authorship

I hereby certify that I have authored this thesis independently and without outside assistance. I have not used any sources or aids other than those specifically cited, and I have clearly indicated all passages taken either verbatim or in spirit from other works. I further certify that the submitted electronic version is identical to the printed copies.

Rostock, 02.03.2026



Muhamad Miari