JOT
JOURNAL OF
OBJECT TECHNOLOGY

Journal of Object Technology | **RESEARCH ARTICLE**

Incremental Task-Driven Composition of Heterogeneous Software Models for Legacy System Maintenance

Azam Mahdpipour, Vera Pantelic, and Sébastien Mosser
McMaster Centre for Software Certification, McMaster University, Canada

ABSTRACT

Maintaining large-scale legacy software systems is complex and time-consuming. Practitioners must integrate diverse artifacts and data, produced by various specialized tools, in order to gain a better understanding of the system and challenges at hand. This process is typically manual and error-prone. In this paper, we propose a solution to support maintenance activities by incrementally merging heterogeneous software models. We build on the existing framework, *Unified Data Source (UDS)*, based on the Single Source of Truth (SST) paradigm. The framework consists of three layers: *Probe*, *SST*, and *Visualization*. The lightweight Probe layer extracts partial models from heterogeneous artifacts and forwards them to the SST layer, which acts as a composition engine that consistently merges the partial models into a unified and coherent representation. The Visualization layer then enables the creation of customized, task-specific model views derived from this unified model. The feasibility and practical value of the method are demonstrated through a systematic, tool-supported approach that applies proven MDE techniques to a concrete system. We showcase this methodology through two real-world use cases: performance troubleshooting and change impact analysis. Both use cases are successfully executed on *Spring PetClinic*, a widely-used demo project, producing task-specific model views without reconstructing the entire system. These results confirm that task-driven, incremental merging of heterogeneous models is practically achievable, yielding actionable views for practitioners and supporting them in making smarter decisions.

KEYWORDS Model Driven Reverse Engineering (MDRE), Software Maintenance, Single Source of Truth, Heterogeneous Model Composition, Visualization

1. Introduction

Legacy software systems impose heavy economic and operational burdens through high maintenance costs. This is illustrated by the U.S. government spending 80% of its $90 billion 2019 IT budget—roughly $72 billion—simply to maintain aging infrastructure (Assunção et al. 2024).

Effective maintenance and informed decision-making require a clear understanding of the system. One key way to achieve this is by integrating the diverse perspectives provided by tool outputs and artifacts into a unified representation (Martínez-Fernández et al. 2018). However, these data are heterogeneous, and according to Boufares & Ben Salem (2012), such integration leads to inconsistencies, including syntax mismatches, incompatible measurement units, divergent data representations, and semantic conflicts.

Many software maintenance solutions, including reverse engineering methods such as Moose (Nierstrasz et al. 2005) and ModMoose (Anquetil et al. 2020), are limited to source code. Therefore, a reliable and trustworthy solution is required to aggregate, structure, and store heterogeneous data consistently. In addition, comprehensive approaches like Rigi (Kienle & Müller 2010) reconstruct entire systems, generating overwhelming data volumes that are time-consuming and costly.

Another limitation of existing software maintenance solutions is their narrow scope. Many approaches are designed

JOT reference format:
Azam Mahdpipour, Vera Pantelic, and Sébastien Mosser. *Incremental Task-Driven Composition of Heterogeneous Software Models for Legacy System Maintenance*. Journal of Object Technology. Vol. 25, No. 3, 2026.
Licensed under Attribution 4.0 International (CC BY 4.0)
http://dx.doi.org/10.5381/jot.2026.25.3.a11

to work only with specific programming languages, tools, or frameworks. In addition, according to Adriana *et al.*'s systematic review (Cruz et al. 2016), many software visualization tools and techniques are language-specific, IDE-dependent, focus on static analysis, and lack support for runtime behaviour—limiting their use in multi-language projects.

To address these limitations, Bryantsev (2025) proposed and developed a framework that seamlessly integrates a wide range of heterogeneous artifacts, operates independently of programming languages and tools, supports diverse maintenance tasks, and remains flexible by avoiding reliance on a single representation technique or tool (while also supporting partial and incremental analysis). In particular, it integrates heterogeneous data from various software tools into a unified representation based on the Single Source of Truth (SST) paradigm. The SST approach is widely used in data-driven systems to ensure data consistency by centralizing all relevant information in one authoritative repository (Queiroz et al. 2024). We will refer to Bryanstev's framework as the *Universal Data Source* (UDS), which supports the implementation of our contribution.

We aim to validate the practical value of the UDS framework for maintaining legacy software systems. To illustrate the framework's support for maintenance activities, we present two real-world use cases: the first, inspired by our industrial partner's needs, focuses on triaging performance issues during troubleshooting; the second addresses change impact analysis, a crucial maintenance task. Specifically:

- We propose a semi-automated UDS-based approach to streamline bug triaging in performance troubleshooting, reducing manual effort.
- We propose a UDS-supported change impact analysis with stakeholder-specific visualizations in *Gephi*.
- We apply well-known and proven-in-use Model-Driven Engineering techniques to a reverse engineering problem on a concrete system via a practical, systematic, tool-supported, state-of-the-art approach.

The remainder of this paper is organized as follows. Section 2 provides the state of the art. Section 3 introduces the UDS framework. In Section 4, two use cases of the framework's application are presented. Section 5 is devoted to the discussion. Section 6 discusses the threats to validity. Finally, Section 7 concludes the paper and discusses future work.

2. State of the Art

The maintenance of legacy software systems presents significant challenges due to the heterogeneity of artifacts. While Model-Driven Reverse Engineering (MDRE) plays a crucial role in extracting models to support evolution, a recent systematic literature review highlights that the vast majority of existing methods rely solely on static source code analysis (Siala et al. 2024). Integration frameworks such as MECASP (Isnard et al. 2005), the Adoption-Centric framework (Huang et al. 2005), and the model-driven Kernel representation by Saeidi *et al.* (Saeidi et al. 2018) have successfully unified documentation, source code, and configurations. Similarly, graph-based tools like the RSG platform (Buchgeher et al. 2018), jQAssistant (Müller et al. 2018), and GETAVIZ (Baum et al. 2017) have extended this scope to structural and limited behavioural models. However, these static-centric approaches systematically exclude runtime data, leaving a gap in understanding how legacy systems actually behave in production.

To address the challenge of linking diverse artifacts, generic model composition frameworks were introduced. The ATLAS Model Weaver (AMW) (Del Fabro et al. 2005) proposed treating the "links" between models as first-class entities (weaving models), enabling fine-grained interoperability between heterogeneous data sources. Building on this, the Epsilon Merging Language (EML) (Kolovos et al. 2006) formalized the integration process into distinct phases (comparison, conformance, and merging) and introduced rule-based strategies to automate the otherwise labour-intensive manual weaving. While these frameworks provide powerful mechanisms for structural composition, such as merging Platform Independent Models with Platform Dependent Models, they are typically designed for static model merging.

Recent efforts have attempted to incorporate dynamic data to address this limitation, though often within narrow, domain-specific boundaries. For instance, Anaximander (Mosser et al. 2020) uses lightweight probes to refine microservice architectures, while Concoction (Wang et al. 2024) combines static graphs with symbolic execution traces for vulnerability detection. In the realm of automated repair and review, extensions to ExplorViz (Krause-Glau et al. 2024) and the LLM-based DEVLoRe framework (Feng et al. 2024) merge static analysis with runtime traces. Despite their utility, these solutions are specialized to particular tasks or architectures and do not provide a generalized mechanism for legacy system maintenance.

Among the approaches surveyed, only one attempts to adapt generic model composition directly for execution traces within a generic framework. The work by Béziers la Fosse *et al.* (Béziers la Fosse et al. 2018) extends MoDisco (Brunelière et al. 2014) by injecting dynamic traces to support test impact analysis (Siala et al. 2024). However, this approach revealed critical scalability limitations inherent to legacy system analysis. Specifically, the reliance on the XMI persistence layer caused memory exhaustion by requiring the entire model to be loaded, preventing partial analysis. Furthermore, the instrumentation phase was computationally expensive due to full-source-code parsing, and the scope of artifacts beyond static code was strictly limited to execution traces.

Therefore, a gap remains for a language- and tool-independent method that consistently aggregates static, dynamic, and evolutionary data while providing incremental, task-specific analysis and custom visualizations that support different maintenance activities. This paper addresses this gap by showing how to use the UDS framework in practice.

3. The UDS-Based Model Composition

To address the challenges of heterogeneous data integration, the SST paradigm is a foundational principle in data management. It emphasizes the need to consolidate key information in a sin-

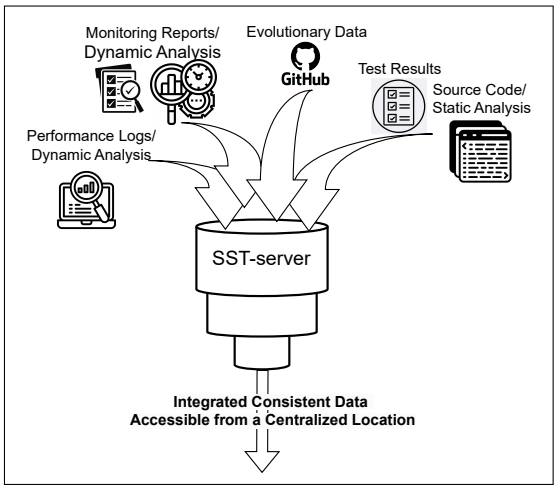


Figure 1 SST paradigm in software maintenance context

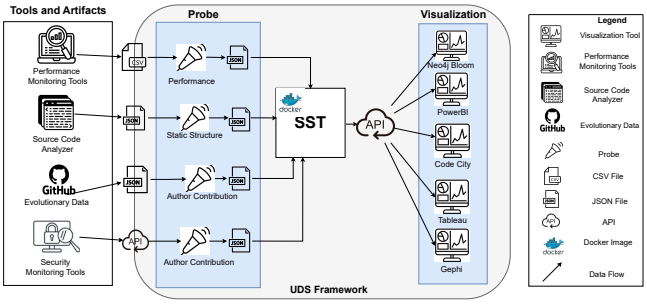


Figure 2 UDS Architecture

gle, authoritative location to guarantee *consistency* (Holzmann 2015).

Figure 1 illustrates how Bryantsev (2025) applies the SST paradigm in the context of software maintenance: it suggests consistently integrating heterogeneous data, artifacts, and partial models produced by diverse tools to provide a centralized unified model, accessible to authorized people. This paradigm serves as the backbone of our solution, specifically driving the separation of concerns within the system. Consequently, the UDS architecture is divided into three conceptual layers: Probe, Visualization, and the SST layer, as shown in Figure 2 and described in the following sections.

3.1. Probe: Heterogeneous Model Extraction Layer

A probe is a lightweight model extractor that transforms raw maintenance artifacts into a standard format. In model-driven terms, each probe performs a model-to-model transformation from a tool-specific partial model to the intermediate model for SST, enabling task-focused incremental analysis, supporting rapid comprehension, and continuous refinement without ever constructing a complete system model. Probes are fully modular, deterministic, and independent of programming languages and tools. They output data in standard formats (CSV/JSON), impose no execution-order constraints, and can be easily reused or adapted across diverse systems and maintenance scenarios.

3.2. SST: Model Merging Layer

The SST layer automatically integrates heterogeneous models into a single, consistent, unified graph model. It achieves this by incrementally merging the graph fragments (partial models) produced by probes. The merging algorithms rely on node equivalence and graph unions[1], inspired by classical approaches such as the Epsilon Merging Language (Kolovos et al. 2006) or the Adore framework (Mosser & Blay-Fornarino 2013).

3.3. Visualization: Multi-View Layer

The visualization layer enables task-specific model views by allowing practitioners to generate custom visualizations that directly consume the composite model generated by the SST layer. Depending on the maintenance task, some analyses benefit from graph exploration, while others require detailed tables, statistical summaries, or charts; since no single tool covers all needs, this design prioritizes customized solutions over generic ones by supporting flexible combinations of tools and custom visualizations to effectively address maintenance challenges.

3.4. Implementation

This research's contributions rely on an existing SST implementation, so we are focusing here on the probe and visualization layers, highlighted in blue boxes in Figure 2, which are described in detail in the next section. For SST, we used the previously mentioned implementation by (Bryantsev 2025), and its Docker image is available at the official `acedesign` repository on Docker Hub[2]. It comprises an SST server backed by the Neo4j[3] graph database. Neo4j's native graph storage, schemaless design, and expressive Cypher query language make it particularly well-suited for managing the highly interconnected and evolving data typical of software maintenance tasks.

According to this implementation, the SST layer operates as a web server handling HTTP requests for probe registration and data uploads. To ensure consistency, it enforces a *Type System* in which probes must first register a schema that defines their node types, fields, and relationships. Incoming data must follow the registered strict JSON format containing node and edge lists (see Listing 1), which the SST maps to a graph model.

Model composition relies on a mathematical *Graph Merging* operation. Using the "Type Equivalence" rules defined during registration (e.g., matching fully qualified method names), the system identifies and merges identical nodes from different tools into a single entity within the *Global Type Schema*. This merging process is associative and commutative, ensuring the final graph model is consistent regardless of the input order. A strict validation pipeline rejects any model that conflicts with the schema to prevent corruption. At this stage of the implementation, direct access to the Neo4j database is available, allowing any compatible tool to be used for querying and exploring the graph data.

In the following section, we validate the practicality and credibility of our approach in real-world applications via two

[1] A complete description of the merging mechanism is out of the scope of this paper; the interested reader can consult Bryantsev (2025) for more information.
[2] https://hub.docker.com/r/acedesign/sst
[3] https://neo4j.com

use cases.

4. Use Cases: Validating UDS in Practice

We assess the practicality of the UDS framework with the help of two detailed real-world use cases, each composing partial models to answer a specific maintenance question.

The selection of use cases was driven by a strategic necessity to bridge immediate industrial needs with fundamental maintenance challenges. Consequently, the first use case (Triaging Performance Troubleshooting) stems directly from an industrial partner's needs, while the second targets a classical problem: Change Impact Analysis (CIA). We select CIA because, as noted in Badri et al. (2015), understanding software and evaluating the effects of changes are widely recognized as the most critical activities in software maintenance.

We choose the *Spring PetClinic* application as the demonstration project because it serves as both a recognized academic benchmark and a realistic proxy for long-running systems. PetClinic exhibits properties that make it suitable for software maintenance research. First, the application has existed since 2013, offering a long evolutionary history. Second, it has a diverse codebase due to the involvement of many developers. Its codebase comprises 23,480 total lines across 114 files in a single-module monolith. The backend contains 2,083 lines of Java within 47 classes and 10 packages. Frontend assets make up the bulk of the project with 9,224 lines of SVG and 8,480 lines of CSS, while configuration files account for 1,097 lines. It is publicly available as open-source code on GitHub[4].

4.1. UC1: Triaging Performance Troubleshooting (TPT)

Triaging is a critical initial step in the software performance troubleshooting process, where reported bugs or issues are assessed and assigned to appropriate developers or teams (Aktas & Yilmaz 2020). Traditionally, issues are assigned manually by project managers by reviewing the issue's summary and description to select the most suitable developer based on expertise and responsibilities (a task that is time-consuming and prone to errors such as incorrect assignments (Oliveira et al. 2021)).

4.1.1. How UDS Supports TPT Process. The primary goal during triage is to identify the individuals/teams best equipped to effectively address the reported issue. Based on the bug life cycle described in Saha et al. (2015) and the bug triage steps outlined in Akila et al. (2015); Anvik et al. (2006); Begum & Dittrich (2024), we illustrate the TPT process using the UDS in Figure 3. The overall workflow follows the structure reported in these references; however, the manual issue investigation step has been replaced with an automated step (highlighted in blue), which involves running the appropriate UDS probes. This automated step enables the triaging team to select and execute probes to collect the required performance metrics based on the specific characteristics of the reported issue. Using these probes and associated visualizations, the team can make informed decisions to identify the most suitable individuals for issue resolution.

[4] https://github.com/spring-projects/spring-petclinic

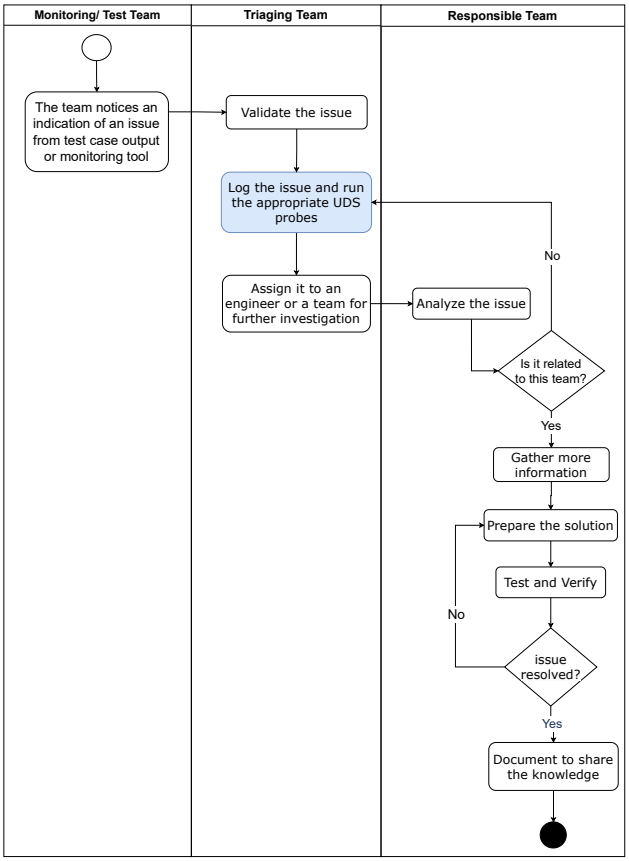


Figure 3 Issue troubleshooting workflow using UDS

According to this workflow, incorrect assignments often lead to reassignments once the assignee realizes the issue is outside their scope, significantly increasing the triage time and cost (Sarkar et al. 2019). Therefore, accurate initial assignments and automated triage are essential to improve efficiency and reduce errors in bug resolution, ultimately enabling effective software maintenance (Xia et al. 2017). Accordingly, research has shifted from labour-intensive and error-prone manual practices toward automated solutions (Sarkar et al. 2019; Zhang et al. 2017).

To elaborate on the activities within the blue step, we consider a concrete example of a high response time issue arising from newly added or recently modified code segments. After analyzing the use case, the developer identifies the required data. Since the issue concerns high response time, three partial models of the system are needed: the methods involved, the locations of these methods in the source code, and the developers associated with changes of the relevant code.

With this analysis, the procedure performed within the blue step comprises the following steps.

1. **Reproduce the Issue:** An unusually high response time of specific software components reported by performance monitoring tools or test cases is validated by developers.

2. **Identify Slow Methods:** To determine which parts of

the software are responsible for the issue, the developer needs to identify methods whose execution times exceed a predefined threshold; such methods are referred to as 'slow methods.' By running the *performance* probe, the required performance metrics, which are measured and collected by a monitoring tool, are extracted and sent to the SST server.

3. **Locate Slow Methods:** Once the 'slow methods' are identified, it is necessary to pinpoint their exact location within the structure of the problematic code. Running the *code structure* probe examines the source code and sends the code structure to the SST server, where it integrates the code structure results with the performance data from the previous step. This allows the precise location of the relevant methods within the source code to be determined.

4. **Find Authorship:** The developers that are best suited to investigate the issue can be identified by analyzing developers' contributions. Such contributions are recognized through version control system analysis, which can be performed by running the *code ownership* probe. Developers who have made the most contributions to the slow methods or who authored the most recent changes are typically the best candidates to resolve the issue.

5. **Assignment:** These developers are then designated as the key personnel responsible for investigating and resolving the performance issue within the respective methods because they are the most familiar with the affected code.

4.1.2. TPT Model.

The three partial models (i.e., performance, code structure, and code authorship) are generated by probes that collect data from specific sources and transmit the results to the SST server in a standard format (JSON) to be integrated. Unifying these models produces the system model required to resolve the TPT issue.

Performance Probe. This probe is implemented as a Python script that extracts runtime data during program execution, acquired through a monitoring or profiling tool; we opt to use *VisualVM*[5]. The script takes a CSV file from VisualVM and transforms it into a structured JSON file containing methods and key performance metrics as shown in Listing 1.

```
{
  "probeName": "Performance_VisualVM",
  "nodes": [
    {
      "type": "Method",
      "fullName": "org.springframework.samples
    .petclinic.owner.Owner.addPet(org.
    springframework.samples.petclinic.owner.
    Pet)"
    },
    {
      "SelfTime": 0.023,
      "SelfTimeCPU": 0.022,
      "TotalTime": 0.044,
      "TotalTimeCPU": 0.045,
      "Invocations": 3,
      "IOOperationTime": 0.001,
      "OutgoingMethodCallTime":
    0.0209999999999999,
      "type": "Performance",
      "uid": "fa80a76e-237a-4c51-b4e6-8
    ce8359af4c5"
    }
  ],
  "edges": [
    {
      "relationName": "PERFORMS",
      "from": {
        "nodeType": "Method",
        "propertyName": "fullName",
        "propertyValue": "org.springframework.
    samples.petclinic.owner.Owner.addPet(org.
    springframework.samples.petclinic.owner.
    Pet)"
      },
      "to": {
        "nodeType": "Performance",
        "propertyName": "uid",
        "propertyValue": "fa80a76e-237a-4c51-
    b4e6-8ce8359af4c5"
      }
    }
  ]
}
```

Listing 1 A JSON example showing the structure of actual graph data

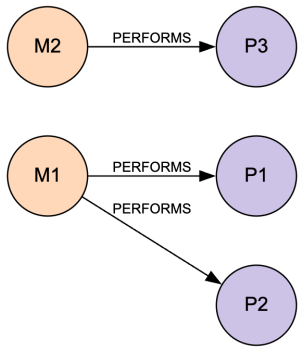
Upon receiving this file, the SST creates corresponding `Method` and `Performance` nodes linked by a `PERFORMS` relationship, as shown in Figure 4a. This partial model captures only the performance-related aspect of the system, providing targeted insight into execution bottlenecks. Figure 4d shows the legend for all graphs examples of this use case.

Code Structure Probe. To locate slow methods and extract structural information from the application's source code, we use Rascal[6], a domain-specific language for static code analysis. The probe takes the source code directory as input and extracts the code structure. Executing the probe generates `File`, `Package`, `Class`, and `Method` nodes, connected by `INCLUDES` (hierarchical) and `INVOKES` (method call) relationships, as shown in Figure 4c. SST automatically creates or merges these nodes with those produced by the performance probe, enabling a unified model that combines code structure and performance metrics.

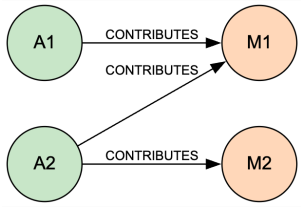
Code Ownership Probe. Finally, to identify contributors to specific methods, evolutionary data is extracted from the version control system (e.g., GitHub). The code ownership probe provides commit information, such as lines of code contributed by each developer and the timing of recent changes. Upon receiving the data, the server creates or merges the corresponding `Method` and `Author` nodes and establishes `CONTRIBUTES` relationships, as shown in Figure 4b. The historical data extracted

[5] https://visualvm.github.io

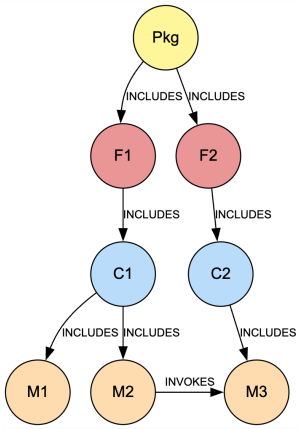
[6] https://www.rascal-mpl.org/



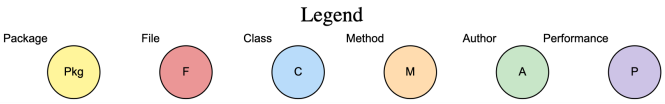
(a) Example of performance model



(b) Example of author contribution model

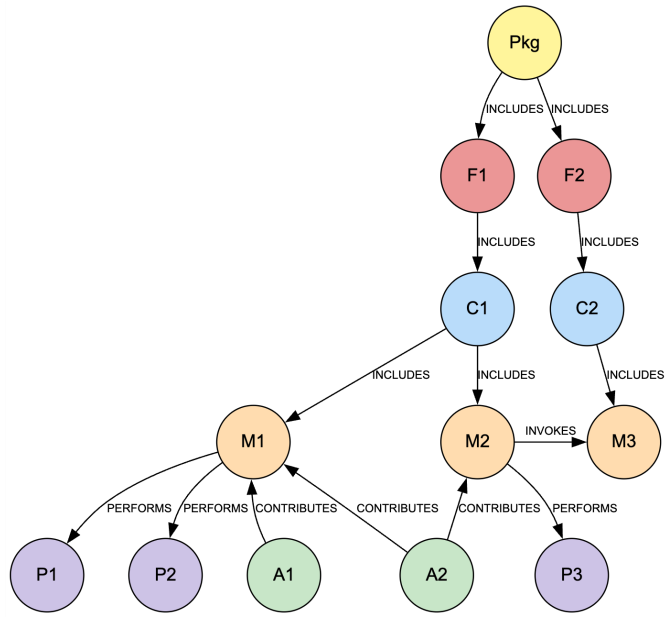


(c) Example of code structure model

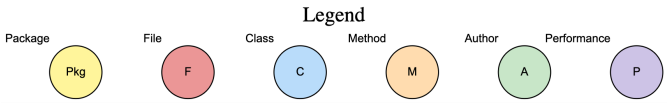


(d) Legend

Figure 4 Graphs produced as a result of each probe



(a) Merged global graph representing the integrated system model



(b) Legend

Figure 5 Final graph model

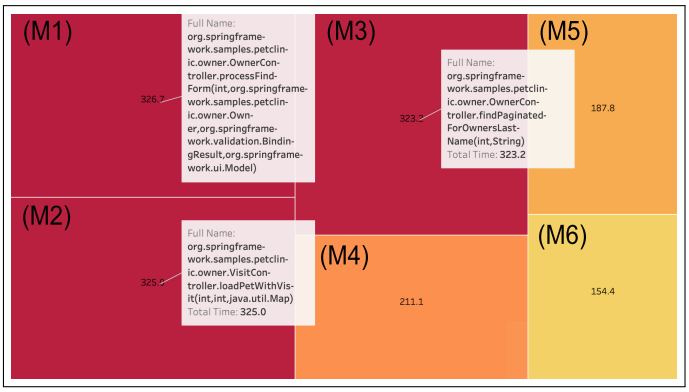


Figure 6 Treemap of methods whose response time exceeds 120 ms. Each rectangle represents a method (M1–M6), defined in Figure 8; the value inside shows response time (ms), and darker colours indicate longer execution time.

from Git (including the number of lines contributed and their timing) is stored as properties of the `CONTRIBUTES` relationship.

4.2. Analysis and Results

Once these heterogeneous partial models are integrated, the resulting graph is shown in Figure 5a, while Figure 5b shows the legend for this graph. The final graph includes 354 nodes and 543 edges.

This solution enables the generation of tailored visualizations for specific needs. For visualization of the TPT use case, we use *Tableau*[7] and the following views.

Figure 6 shows a treemap[8] to display the methods whose total response times exceed 120 ms (a predefined threshold to define slow methods), highlighting six slow methods (M1–M6) with total times of 326.7, 325.0, 323.2, 211.1, 187.8, and 154.4 milliseconds. The M1–M6 identifiers correspond to the full method names listed in Figure 8.

The stacked bar chart in Figure 7 shows the contributors to each slow method, enabling identification of the developers responsible for them. Methods are annotated as M1–M6, corresponding to the slow methods listed in Figure 8.

Finally, Figure 8 provides the answer to the use case question "*Who is the appropriate developer to fix the reported bug?*" It presents data on top and the latest developers associated with each slow method.

For example, it reveals that `Simrin051` made the most contributions to the `org.springframework.samples.petclinic.vet.owner.PetController.PopulatePetTypes()` method, while `Dave Syer` was the last developer to modify it.

The method of identifying appropriate developers based on recent and significant contributions to slow methods is merely one possible method of choosing a developer based on our expe-

[7] https://www.tableau.com/
[8] https://www.tableau.com/chart/what-is-treemap

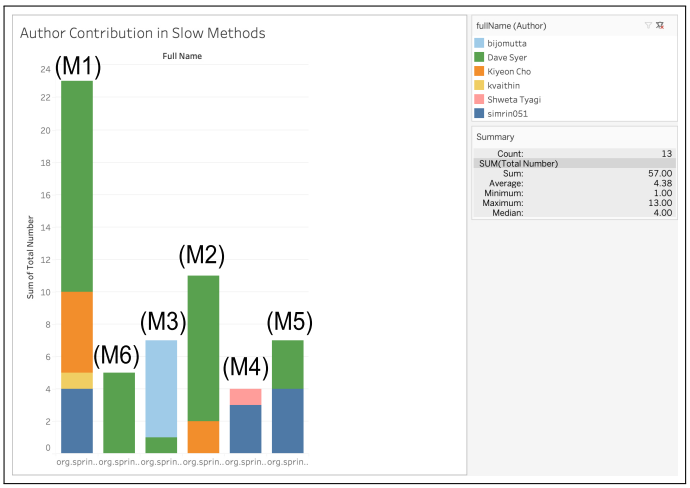


Figure 7 Main contributors to the slow methods. Developer contributions are shown for methods M1–M6.

No.	Method Name	Top Contributor	Latest Contributor
(M1)	org.springframework.samples.petclinic.owner.OwnerController.processFindForm()	Dave Syer	Dave Syer
(M2)	org.springframework.samples.petclinic.owner.VisitController.loadPetWithVisit(int,int,java.util.Map)	Dave Syer	Dave Syer
(M3)	org.springframework.samples.petclinic.owner.OwnerController.findPaginatedForOwnersLastName(int,String)	Bijomutta	Bijomutta
(M4)	org.springframework.samples.petclinic.owner.OwnerController.findOwner(Integer)	simrin051	Shweta Tyagi
(M5)	org.springframework.samples.petclinic.owner.PetController.populatePetTypes()	simrin051	Dave Syer
(M6)	org.springframework.samples.petclinic.owner.PetTypeFormatter.parse(String,java.util.Locale)	Dave Syer	Dave Syer

Figure 8 Preferred developers for each slow method (M1-M6), ordered by execution time.

rience with an industrial partner. If a company chooses to redefine the notion of the most appropriate developer, our framework remains applicable: it would simply require using a new probe to collect the data necessitated by the new definition. Additional data sources (e.g., JIRA/Bugzilla history, bug-resolution patterns, developer expertise, and availability (Jahanshahi & Cevik 2022) can be integrated via a new probe. Machine learning or rule-based ranking can be used to recommend the most qualified developers, reducing triaging time and improving assignment accuracy as the project evolves.

4.3. UC2: Multi-Level CIA

Software change requests frequently cause ripple effects, i.e., unintended side effects that propagate through dependencies such as control and data flows (Yau et al. 1800). According to Lehnert's CIA taxonomy (Lehnert 2011), ripple effects are classified as direct (immediate impact on entities with explicit dependencies, e.g., direct calls or variable uses) or indirect (cascading propagation through subsequently affected entities).

Therefore, change requests often affect multiple system parts, introduce inconsistencies, and complicate cost/time estimation for managers, precise change localization for developers, and efficient test case selection for test engineers (Sun et al. 2012). CIA addresses these challenges by identifying the scope of a change, thereby reducing manual dependency tracing, avoiding unnecessary modifications, preventing costly rework, and ultimately lowering overall maintenance effort and budget.

Change requests can occur at different levels, such as requirements, architecture, configuration, or code level. Our focus in this section is on code-level and configuration changes, examining how UDS supports fine-grained operations by answering two key questions: "If I change a method's signature, what else might break or need fixing?" and "If I update an external library, what else might be affected?"

4.3.1. How UDS Supports CIA.

Figure 9 illustrates the CIA process using UDS. While the original CIA process is described in (Dhamija & Sikka 2019), the modifications introduced by UDS are shown in bold in the two blue steps. The process starts with receiving the change request, followed by creating a change set that includes all client-requested modifications. The maintenance team then reviews these change requests and identifies the locations in the source code where the corresponding functionality is implemented. All entities that must be changed initially to fulfill the change request are collected in a set called the *Starting Impact Set (SIS)*.

From there, the CIA then estimates the ripple effects. The SIS, combined with additional entities likely to be affected by ripple effects from changes in the SIS entities, forms the *Estimated Impact Set (EIS)* (Dhamija & Sikka 2019).

To illustrate the steps in the two blue boxes, we provide two concrete examples:

1. A change is requested to the signature of the method m, which is `org.springframework.samples.petclinic.owner.Owner.addPet`. We demonstrate how the UDS framework assists three different roles: developers (by identifying SIS_d and EIS_d), test engineers (by SIS_t and EIS_t) and managers (by SIS_m and EIS_m).

2. One of the project's external libraries is requested to change, and the goal of the procedure is to support developers in making informed decisions (by SIS_l and EIS_l). For visualization purposes of these use cases, we use *Gephi*[9]. Since the names are long and displaying them as node labels reduces readability, we instead use node IDs in the graphs.

   The final graph for the CIA use case includes 284 nodes and 630 edges.

4.3.2. CIA Model – For Developers

Required Probes. Since the change involves a method signature, it is first necessary to determine the precise location of the method within the software's structural hierarchy. This is

[9] https://gephi.org/

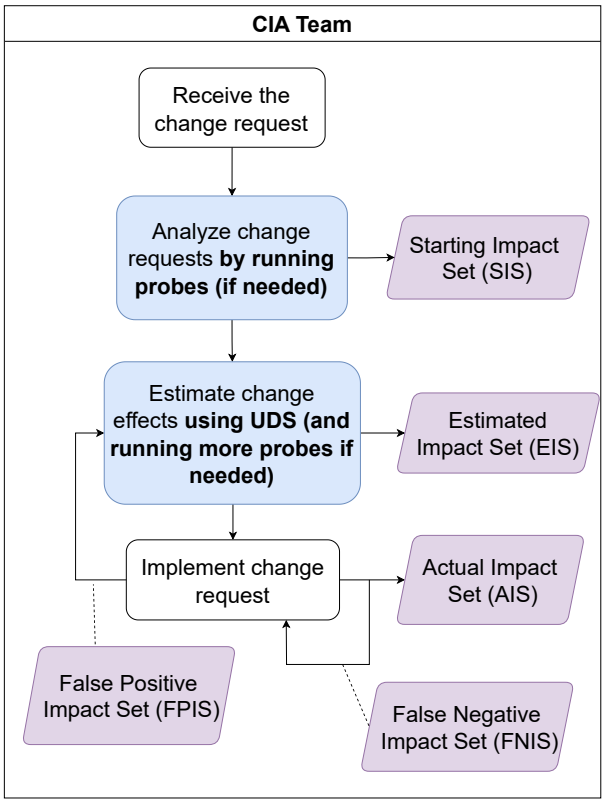


Figure 9 The CIA process using UDS

achieved using the *code structure probe*, which returns the full code structure, including all packages, classes, and methods (reusing from the TPT use case). Second, because modifying the method signature may affect the locations where it is invoked, it is essential to identify these callers. To do so, we rely on call-dependency relationships between methods. These dependencies are obtained by executing the *Static Call Stack* probe (which uses Rascal for static code analysis) and the *Dynamic Call Stack* probe (which uses the VisualVM profiler to collect runtime method calls). These probes extract both static and dynamic caller/callee relationships (including test methods), represent them as nodes with directed `INVOKES` edges, and transmit the resulting data to the SST server for merging with the code structure model.

Procedure and Results. With the required data identified, the procedure is as follows:

1. **Receive the change request:** The process begins with the requested change of the signature of the method m (ID 73).

2. **Analyze change request:** To analyze the change request, the developer runs the code structure probe; the structural details of the code are extracted and sent to the SST server. Therefore, we can identify the location of the method m in the result graph model in Gephi. For a method signature change, the SIS for this case, SIS_d, is defined as the specific method that must be directly modified to satisfy the change, so SIS_d is $\{m\}$.

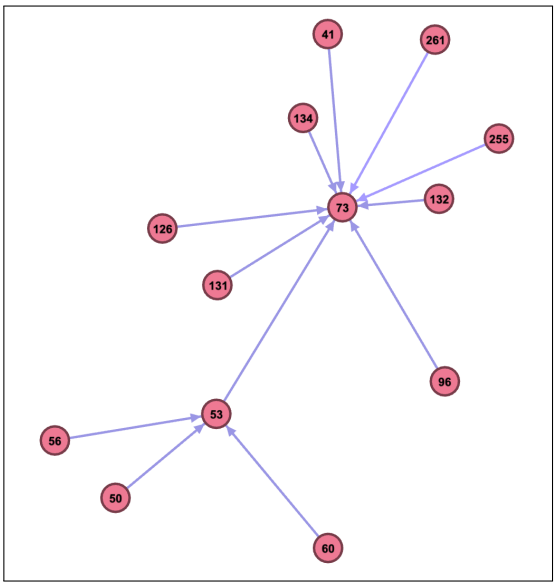


Figure 10 The EIS_d for changing the method 73

3. **Estimate change effects**: The next step is to determine all methods that depend on m. Running the static call stack and dynamic call stack probes generates the method call graph. To construct a complete graph, we executed various tests to cover all functionalities of the PetClinic project. In practice, however, such coverage would be achieved gradually through user interactions. According to Lehnert (2011), static and dynamic impact sets can be combined in three main ways: union (`Static` ∪ `Dynamic`), intersection (`Static` ∩ `Dynamic`), or weighted/ranked combination, where dependencies are scored and prioritized by execution frequency or likelihood instead of being treated equally. We have chosen union for this example to ensure no potential impacts are missed. To construct EIS_d, by traversing the unified call graph backward from the changed element of m, the system produces the caller set. First-order callers are direct callers (methods with IDs 131, 126, 134, 41, 261, 255, 132, 96, and 53), and higher-order callers are indirect callers (methods with IDs 50, 56, and 60), which is shown in Figure 10. EIS_d is formed by combining SIS_d with all its direct and indirect callers. To obtain a list of the names and additional information about these methods, Gephi provides access to the data for the nodes and edges in a tabular format, as shown in Figure 11. The `org.springframework.samples.petclinic` package prefix is omitted from the method names for clarity in Figures 11, 14, and 17.

4.3.3. CIA Model – For Test Engineers. Test engineers seek to minimize the number of test cases in response to changes, ensuring testing remains efficient and effective. We can apply a similar process to assist test engineers by computing EIS_t from the testing perspective. This analysis can be performed in

Id	method_fullname
41	.service.ClinicServiceTests.shouldInsertPetIntoDatabaseAndGenerateId()
50	.owner.OwnerControllerTests.setup()
53	.owner.OwnerControllerTests.george()
56	.owner.OwnerControllerTests.testProcessFindFormByLastName()
60	.owner.OwnerControllerTests.testProcessFindFormNoOwnersFound()
73	.owner.Owner.addPet(org.springframework.samples.petclinic.owner.Pet)
96	.owner.PetControllerTests.setup()
126	.owner.VisitControllerTests.init()
131	.owner.PetController.initCreationForm(org.springframework.samples.petclinic.owner.Owner
132	.owner.PetController.processCreationForm(org.springframework.samples.petclinic.owner...
134	.owner.PetController.processUpdateForm(org.springframework.samples.petclinic.owner...
255	.owner.PetController.initCreationForm(org.springframework.samples.petclinic.owner.Owner
261	.owner.PetController.processCreationForm(org.springframework.samples.petclinic.owner..

Figure 11 List of all methods in EIS_d

two ways. Incrementally, by reusing the previously computed EIS_d and extending it with test-specific dependencies, or independently, from scratch, in which case three probes of code structure, static and dynamic call stacks must be executed first since identifying EIS_t requires knowledge of EIS_d.

Required Probes. In addition to the code structure, static call, and dynamic call probes, a test perspective is required to validate changes and detect potential regressions early. The *test dependency* probe addresses this by using the Spoon tool[10] to analyze test files, extract unit tests, and identify the tested production methods. It models both test cases and tested methods as nodes, with directed `TESTS` edges from each test case to the method it exercises. Test cases are also represented as regular method nodes, since they are implemented as methods.

Procedure and Results. With the required data identified, the procedure is as follows:

1. **Receive the change request:** This step follows the same initial setup as the developer-focused process, starting with method m (ID 73) whose signature is to be modified.

2. **Analyze change request:** As the developer continues the analysis after the developer-focused procedure, the location of m is already known. Test engineers need to determine which test cases (e.g., unit tests) are related to m, therefore, SIS_t includes the test cases directly associated with m. To visualize the results of this step, we filter the test cases that test method ID 73, i.e., nodes with a direct `TESTS` edge to method 73 in the unified graph model. This SIS_t includes methods with IDs 41, 126, 53, and 96, which are shown in Figure 12.

3. **Estimate change effects**: At this stage, engineers must also verify the ripple effects of the change in m—that is, they need to test all methods belonging to EIS_d. By traversing the dependency graph and selecting all test cases that are connected to any method in EIS_d, the EIS_t will be identified, which is shown in Figure 13. The names of these methods are shown in Figure 14.

4.3.4. CIA Model – For Managers. To support managers in planning and resource allocation, we identify which developers (authors) are likely to be affected by the change or may need

[10] https://spoon.gforge.inria.fr/

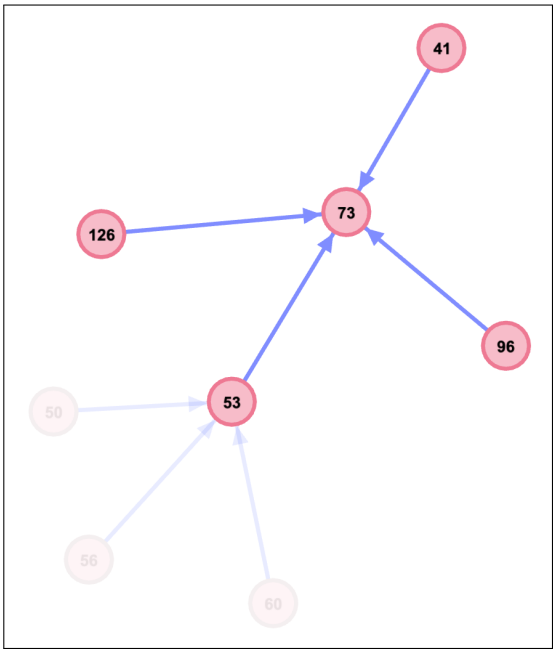


Figure 12 The SIS_t for test engineers

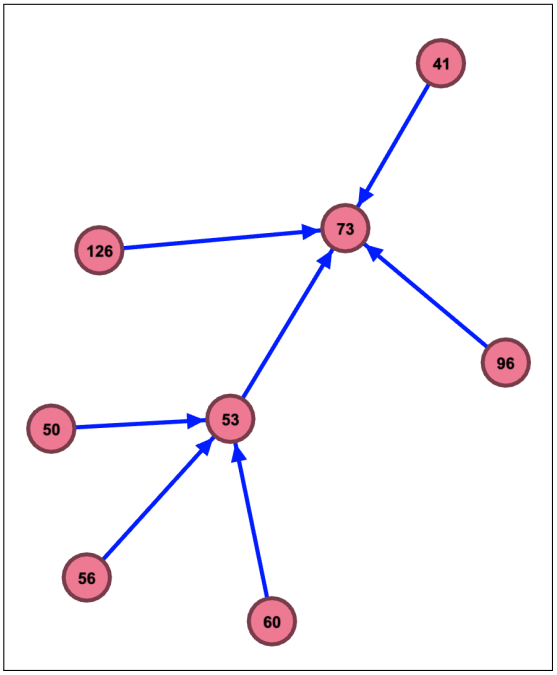


Figure 13 The EIS_t for test engineers

Id	method_fullname
41	.service.ClinicServiceTests.shouldInsertPetIntoDatabaseAndGenerateId()
50	.owner.OwnerControllerTests.setup()
53	.owner.OwnerControllerTests.george()
56	.owner.OwnerControllerTests.testProcessFindFormByLastName()
60	.owner.OwnerControllerTests.testProcessFindFormNoOwnerFound()
73	.owner.Owner.addPet(org.springframework.samples.petclinic.owner.Pet)
96	.owner.PetControllerTests.setup()
126	.owner.VisitControllerTests.init()

Figure 14 List of all methods in EIS_t

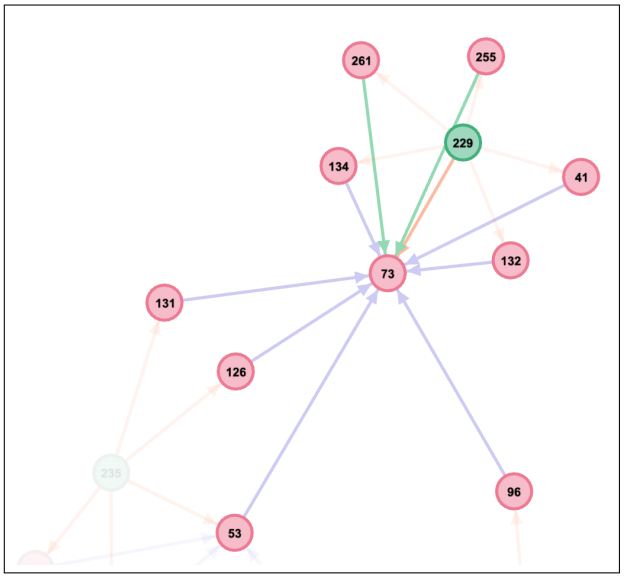


Figure 15 The SIS_m

to be involved in its implementation or review. This procedure follows the same steps as previous processes and depends on the previously computed EIS_d.

Required Probes. In addition to the code structure, static call, and dynamic call probes, it is necessary to identify the developer(s) responsible for the changed method. The *code ownership* probe analyzes the GitHub repository, establishing `Author` and `Method` nodes connected by `CONTRIBUTES` relationships (reusing from the TPT use case).

Procedure and Results. With the required data identified, the procedure is as follows:

1. **Receiving the change request:** This step follows the same initial setup as the developer-focused process, starting with method m (ID 73) whose signature is to be modified.
2. **Analyze change request:** By running the code ownership probe, it enriches the unified dependency graph with authorship information. Consequently, SIS_m consists of the developer(s) directly associated with the m. These authors are immediately visible on the unified graph shown via the visualization tool, as shown in Figure 15.
3. **Estimate change effects**: At this stage, we need to identify the developers responsible for all methods in EIS_d to provide managers with a complete picture of the required expertise and potential workload impact. By traversing the unified dependency graph and collecting all authors linked to the methods belonging to EIS_d, the resulting set of affected developers, together with the authors in SIS_m, as EIS_m, are shown in Figure 16. The data associated with the list of authors, which includes `Dave Syer`, `Bijomutta`, and `Shweta Tyagi`, is shown in Figure 17.

4.3.5. CIA Model (External Libraries Case) – For Developers. Unlike the previous use case, which focused on method-level analysis, the objective here is to demonstrate that the level

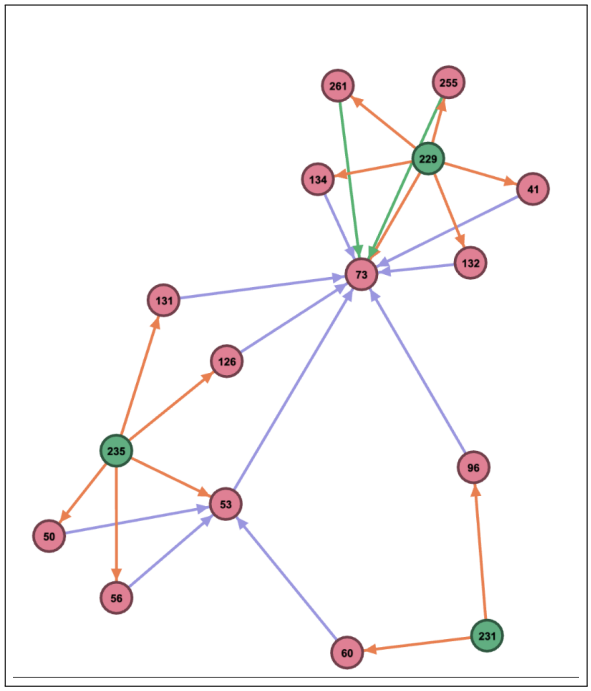


Figure 16 The EIS_m

Id	labels	method_fullname	author_fullname
41	[Method]	.service.ClinicServiceTests.shouldInsertPetIntoDatabaseAn	
50	[Method]	.owner.OwnerControllerTests.setup()	
53	[Method]	.owner.OwnerControllerTests.george()	
56	[Method]	.owner.OwnerControllerTests.testProcessFindFormByLastN	
60	[Method]	.owner.OwnerControllerTests.testProcessFindFormNoOwn	
73	[Method]	.owner.Owner.addPet(org.springframework.samples.petclinic	
96	[Method]	.owner.PetControllerTests.setup()	
126	[Method]	.owner.VisitControllerTests.init()	
131	[Method]	.owner.PetController.initCreationForm(org.springframework.	
132	[Method]	.owner.PetController.processCreationForm(org.springframew	
134	[Method]	.owner.PetController.processUpdateForm(org.springframewo	
229	[Author]		Dave Syer
231	[Author]		Shweta Tyagi
235	[Author]		bijomutta
255	[Method]	.owner.PetController.initCreationForm(org.springframework.	
261	[Method]	.owner.PetController.processCreationForm(org.springframew	

Figure 17 Information on potentially affected authors

of detail in CIA can vary depending on the specific needs and objectives. In this second use case, we shift the granularity to file-level analysis. While the approach can be extended to analyze potentially affected test cases or involved authors, here we focus on the developers' perspectives.

This use case concerns modifying one of the external libraries used in the project. This situation may arise when we want to replace the library with an alternative or, for example, update its version, so we will show how to get SIS_l and EIS_l.

Required Probes In addition to the code structure probe that provides the foundational structure of the project, we need to examine the project's build configuration and identify dependencies on external libraries. The *external dependencies probe* analyzes the Project Object Model (POM) file and Java files, adding them as `File` and `Library` nodes, and connects them with `DEPENDS` relationships.

Procedure and Results With the required data identified, the procedure is as follows:

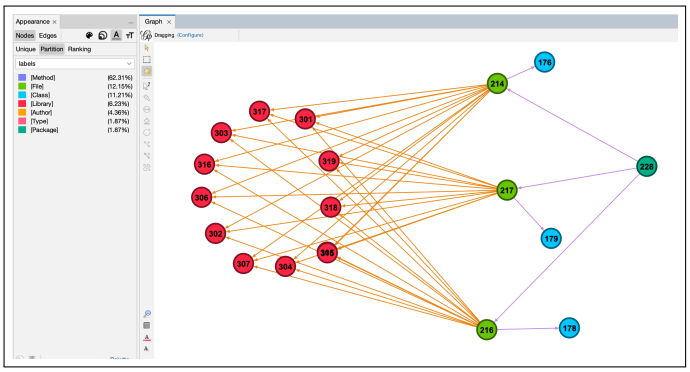


Figure 18 The EIS_l and affected files, which imports Library l and corresponding classes and package

1. **Receiving the change request:** The process begins with a change request that identifies the library l (ID 319) to be modified.

2. **Analyze change request:** The developer begins by analyzing the change request. In this case, the requested change affects only an external library. Consequently, SIS_l consists solely of the library l.

3. **Estimate change effects**: To determine the actual impact of l, the developer runs the *external dependencies probe* and identifies all source files that directly import l, forming the direct ripple effect, files with ID 214, 216, and 217. For the indirect impact, the developer reuses or executes the *code structure* probe. By combining both probe results in the unified graph, which is done by the SST server, we can traverse from the directly affected files to reveal all dependent classes and packages. Consequently, (EIS_l) consists of SIS_l, all source files that are directly referenced l and all packages and classes that depend on those directly affected files, as shown in Figure 18. It also shows the libraries in the affected files, but we are focusing on library 319. The corresponding fully qualified names of the elements are listed in Figure 19.

We demonstrated how UDS supports CIA at multiple levels (method and file levels) on Spring PetClinic through two representative use cases visualized with Gephi. The method-signature change reveals potentially affected methods, test cases, and responsible developers, while the external-library update identifies impacted files, classes, and packages. Together, these cases illustrate that UDS delivers flexible, multi-level, task-specific views that enable informed decision-making for developers, test engineers, and managers alike.

Regarding the performance of the use cases in terms of the query execution time, the average query time (including running the probes to collect the data and executing the queries to generate the visualizations) was 8 seconds across the two scenarios, with no query exceeding 10 seconds. The experiments were conducted on a laptop equipped with an Apple M2 processor and 24 GB of memory, running macOS Tahoe.

5. Discussion

The results presented in this paper confirm the feasibility of the proposed solution and bring to light several important points:

Effectiveness of Task-specific Views Our approach follows an incremental strategy, employing targeted probes to construct task-specific models of the PetClinic system that were sufficient to address specific use-case requirements without requiring a full system-wide analysis. Flexible and incremental model integration allows these probes to be added progressively, enabling the unified model to evolve in response to emerging analysis needs without full reconstruction.

Technology Independence Regarding technology independence, we implemented the test dependencies probe in Java and the remaining probes in Python, while leveraging UDS flexibility to customize visualizations, using Tableau for the TPT case and Gephi for the CIA analysis due to its superior graph-centric features for exploring and presenting relationships.

Challenges in Determining Relevant Data Selecting the right probes is more like a classic signal-to-noise or, equivalently, a needle-in-a-haystack challenge. Without prior knowledge of the maintenance context and the team's concrete goals, it is easy to either drown the useful signal in overwhelming noise by applying too many probes (introducing unnecessary complexity and overhead) or to miss the needle entirely by using too few probes and overlooking the critical hotspots that genuinely matter. Effective probe selection, therefore, requires informed judgment to maximize the signal-to-noise ratio and ensure that the real problems are found rather than remaining hidden in the haystack.

Reusable Probes We reused the same probes (code structure and code ownership) across two use cases without modification, as they are reusable across multiple scenarios and software systems.

Inconsistency Handling in Model Composition Given that we aggregate heterogeneous models, two situations of inconsistencies can arise when multiple probes refer to the same entity. First, different models may report different values for the same property (e.g., a method's response time) because measurements were taken at different times or under various conditions. Both values are correct within their respective contexts; this is expected in a dynamic system and does not represent a conflict. We therefore attach all such observations to the proper element. The second situation arises when two models assert contradictory facts about the same element, which cannot be true simultaneously. Under our design assumption that probes are deterministic and accurate, this situation should never occur. If it happens, the SST server detects and reports it to a developer for resolution, while the composite model remains consistent and trustworthy.

Id	labels	class_shortname	file_filename	package_fullname	library_uid
176	[Class]	MysqlTestApplication			
178	[Class]	MySqlIntegrationTests			
179	[Class]	PostgresIntegrationTests			
214	[File]		/src/test/java/org/springframework/samples/petclinic/MysqlTes...		
216	[File]		/src/test/java/org/springframework/samples/petclinic/MySqlInt...		
217	[File]		/src/test/java/org/springframework/samples/petclinic/PostgresI...		
228	[Package]			org.springframework.samples.petcl...	
301	[Library]				org.springframework.boot:spring-...
302	[Library]				org.springframework.boot:spring-...
303	[Library]				org.springframework.boot:spring-...
304	[Library]				org.springframework.boot:spring-...
305	[Library]				org.springframework.boot:spring-...
306	[Library]				org.springframework.boot:spring-...
307	[Library]				org.springframework.boot:spring-...
315	[Library]				org.springframework.boot:spring-...
316	[Library]				org.springframework.boot:spring-...
317	[Library]				org.springframework.boot:spring-...
318	[Library]				org.testcontainers:junit-jupiter:UN...
319	[Library]				org.testcontainers:mysql:UNKNO...

Figure 19 Names of potentially affected classes, packages, and files

6. Threats to Validity

We acknowledge potential threats to the validity of our study as follows.

Generalizability (External Validity) Our validation is limited to two maintenance activities within a single target system. To mitigate this, we employed a strategic selection process, prioritizing use cases that address the most critical practical and theoretical issues in the domain. Furthermore, we selected Spring PetClinic as our target system; it serves as a neutral, standardized benchmark that allows us to simulate the evolution of legacy systems without the constraints often associated with proprietary code. While we acknowledge that PetClinic acts as a proxy, it ensures reproducibility and provides a baseline for comparison. Future work will aim to extend these findings through large-scale validation across diverse industrial architectures.

Configuration Overhead vs. Universality (External Validity) The UDS framework requires initial effort and domain expertise to implement probes and visualizations, which may present a barrier to adoption. However, this is a deliberate trade-off to ensure flexibility and tool independence. We argue that this initial setup effort is necessary to enable a universal heterogeneous model analysis. Furthermore, while the integration of numerous probes may result in a large unified model, this does not necessarily increase engineers' cognitive workload. UDS is designed to support task-specific system models, filtered views, and tailored visualizations that allow engineers to interact with relevant subsets of the model rather than its entirety. We anticipate that this burden will decrease as a community-driven ecosystem is established.

Absence of Baselines (Comparative Validity) We did not perform a quantitative comparison against state-of-the-art tools. A direct benchmark is infeasible because no existing tool provides a unified framework comparable to UDS's diverse capabilities. While specialized tools exist for individual use cases, they lack the flexibility of our approach to accommodate diverse languages, tools, and maintenance tasks, making a direct global comparison impossible. This study, therefore, focuses on establishing practical feasibility. We reserve rigorous performance benchmarking for future work, where we intend to validate the framework on a per-use-case basis against respective specialized tools.

7. Conclusions

This paper demonstrates how a framework based on the SST paradigm can support software maintenance activities in legacy software systems by presenting two classical software maintenance use cases. The proposed approach to leveraging UDS involves collecting relevant models from heterogeneous artifacts using reusable probes, automatically composing them within the SST server into a unified graph model, and leveraging tailored visualizers to address the specific challenges of the use cases. This method enables engineers to perform maintenance analysis incrementally, gradually refining the model as needs evolve.

To make the UDS framework more mature and widely used, we plan to (1) validate it on many more diverse maintenance and evolution activities (e.g., security-vulnerability detection, performance optimization) in additional industrial codebases with long-term case studies; (2) use large language models to automatically suggest and generate probes to lower the adoption barrier for new projects; (3) integrate incremental probe execution into CI/CD pipelines and provide immediate feedback in pull requests and IDEs; (4) create an open source community repository of reusable probes and visualization dashboards; and (5) perform systematic quantitative benchmarks against state-of-the-art approaches and tools. With these steps, we expect UDS to become a practical, widely adopted solution for maintaining large legacy systems.

Acknowledgments

This work is funded by *McMaster Faculty of Engineering*, as well as the *Natural Science and Engineering Research Council* (NSERC) of Canada, thanks to the Discovery Grant program (RGPIN-2020-05791).

References

Akila, V., Zayaraz, G., & Govindasamy, V. (2015). Effective bug triage–a framework. *Procedia Computer Science*, *48*, 114–120. (International Conference on Computer, Communication and Convergence (ICCC 2015)) doi: https://doi.org/10.1016/j.procs.2015.04.159

Aktas, E. U., & Yilmaz, C. (2020). An Exploratory Study on Improving Automated Issue Triage with Attached Screenshots. In *2020 IEEE/ACM 42nd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)* (pp. 292–293).

Anquetil, N., Etien, A., Houekpetodji, M. H., Verhaeghe, B., Ducasse, S., Toullec, C., … Derras, M. (2020). Modular Moose: A new generation of software reverse engineering platform. In *19th International Conference on Software and Systems Reuse* (pp. 119–134). Berlin, Heidelberg: Springer-Verlag. doi: 10.1007/978-3-030-64694-3_8

Anvik, J., Hiew, L., & Murphy, G. C. (2006). Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering* (pp. 361–370). New York, NY, USA: Association for Computing Machinery. doi: 10.1145/1134285.1134336

Assunção, W. K. G., Marchezan, L., Egyed, A., & Ramler, R. (2024). *Contemporary software modernization: Perspectives and challenges to deal with legacy systems.* Retrieved from https://arxiv.org/abs/2407.04017

Badri, L., Badri, M., & Joly, N. (2015, 04). Towards a change impact analysis model for Java programs: An empirical evaluation. *Journal of Software*, *10*, 441–453. doi: 10.17706/jsw.10.4.441-453

Baum, D., Schilbach, J., Kovacs, P., Eisenecker, U., & Müller, R. (2017). GETAVIZ: Generating structural, behavioral, and evolutionary views of software systems for empirical evaluation. In *2017 IEEE Working Conference on Software Visualization (VISSOFT)* (pp. 114–118). doi: 10.1109/VISSOFT.2017.12

Begum, M., & Dittrich, Y. (2024, December). *What happens in triage? An empirical study of bug triage in software product evolution?* (Tech. Rep. No. TR-2024-210). IT University of Copenhagen.

Béziers la Fosse, T., Tisi, M., & Mottu, J.-M. (2018). Injecting execution traces into a model-driven framework for program analysis. In M. Seidl & S. Zschaler (Eds.), *Software technologies: Applications and foundations* (pp. 3–13). Cham: Springer International Publishing.

Boufares, F., & Ben Salem, A. (2012). Heterogeneous data-integration and data quality: Overview of conflicts. In *2012 6th International Conference on Sciences of Electronics, Technologies of Information and Telecommunications (SETIT)* (pp. 867–874). doi: 10.1109/SETIT.2012.6482029

Brunelière, H., Cabot, J., Dupé, G., & Madiot, F. (2014). Modisco: A model driven reverse engineering framework. *Information and Software Technology*, *56*(8), 1012–1032. doi: https://doi.org/10.1016/j.infsof.2014.04.007

Bryantsev, S. (2025). *The Single Source of Truth paradigm as a tool for supporting software maintenance* (Master of Applied Science, McMaster University, Hamilton, Ontario, Canada). Retrieved from https://dsp.mcmaster.ca/bitstream/11375/31578/2/Bryantsev_Stepan_2025_Apr_MASc.pdf (Supervised by Dr. Sébastien Mosser)

Buchgeher, G., Weinreich, R., & Huber, H. (2018). A platform for the automated provisioning of architecture information for large-scale service-oriented software systems. In C. E. Cuesta, D. Garlan, & J. Pérez (Eds.), *Software Architecture* (pp. 203–218). Cham: Springer International Publishing.

Cruz, A., Bastos, C., Afonso, P., & Costa, H. (2016). Software visualization tools and techniques: A systematic review of the literature. In *2016 35th International Conference of the Chilean Computer Science Society (SCCC)* (pp. 1–12). doi: 10.1109/SCCC.2016.7836048

Del Fabro, M. D., Bézivin, J., Jouault, F., Breton, E., & Gueltas, G. (2005, June). AMW: A generic model weaver. In *Premières Journées sur l'Ingénierie Dirigée par les Modèles (IDM'05)* (pp. 1–10). Paris, France.

Dhamija, A., & Sikka, S. (2019, 06). A systematic study of advancements in change impact analysis techniques. *International Journal of Innovative Technology and Exploring Engineering (IJITEE)*, *8*, 435–443.

Feng, Q., Ma, X., Sheng, J., Feng, Z., Song, W., & Liang, P. (2024). Integrating various software artifacts for better LLM-based bug localization and program repair. *ArXiv*, *abs/2412.03905*. doi: 10.48550/arXiv.2412.03905

Holzmann, G. J. (2015). Points of truth. *IEEE Software*, *32*(4), 18–21. doi: 10.1109/MS.2015.103

Huang, S., Tilley, S., VanHilst, M., & Distante, D. (2005). Adoption-centric software maintenance process improvement via information integration. In *13th IEEE International Workshop on Software Technology and Engineering Practice (STEP'05)* (pp. 25–34). doi: 10.1109/STEP.2005.8

Isnard, E., Perez, E., Bercaru, R., Galatescu, A., Florian, V., Conescu, D., … Stanciu, A. (2005). Integration and maintenance of heterogeneous applications and data structures. In T. Yakhno (Ed.), *Advances in Information Systems.* Berlin, Heidelberg: Springer Berlin Heidelberg.

Jahanshahi, H., & Cevik, M. (2022). S-DABT: Schedule and dependency-aware bug triage in open-source bug tracking systems. *Information and Software Technology*, *151*, 107025. doi: https://doi.org/10.1016/j.infsof.2022.107025

Kienle, H. M., & Müller, H. A. (2010, April). Rigi-an environment for software reverse engineering, exploration, visualization, and redocumentation. *Sci. Comput. Program.*, *75*(4), 247–263. doi: 10.1016/j.scico.2009.10.007

Kolovos, D. S., Paige, R. F., & Polack, F. A. C. (2006). Merging models with the Epsilon Merging Language (EML). In O. Nierstrasz, J. Whittle, D. Harel, & G. Reggio (Eds.), *Model Driven Engineering Languages and Systems* (pp. 215–229). Berlin, Heidelberg: Springer Berlin Heidelberg.

Krause-Glau, A., Damerau, L., Hansen, M., & Hasselbring, W. (2024). Visual integration of static and dynamic software analysis in code reviews via software city visualization. In *2024 IEEE Working Conference on Software Visualization (VISSOFT)* (pp. 144–149). doi: 10.1109/VISSOFT64034.2024.00028

Lehnert, S. (2011). *A review of software change impact analy-*

sis. Retrieved from https://api.semanticscholar.org/CorpusID:10622808

Martínez-Fernández, S., Jovanovic, P., Franch, X., & Jedlitschka, A. (2018, August). Towards automated data integration in software analytics. In *Proceedings of the International Workshop on Real-Time Business Intelligence and Analytics* (pp. 1–5). ACM. doi: 10.1145/3242153.3242159

Mosser, S., & Blay-Fornarino, M. (2013). "adore", a logical meta-model supporting business process evolution. *Sci. Comput. Program.*, *78*(8), 1035–1054. Retrieved from https://doi.org/10.1016/j.scico.2012.06.009 doi: 10.1016/j.scico.2012.06.009

Mosser, S., Caissy, J.-P., Juroszek, F., Vouters, F., & Moha, N. (2020). Charting microservices to support services' developers: The Anaximander approach. In E. Kafeza, B. Benatallah, F. Martinelli, H. Hacid, A. Bouguettaya, & H. Motahari (Eds.), *Service-Oriented Computing - 18th International Conference, ICSOC 2020, Dubai, United Arab Emirates, December 14–17, 2020, Proceedings* (Vol. 12571, pp. 36–44). Springer. doi: 10.1007/978-3-030-65310-1_3

Müller, R., Mahler, D., Hunger, M., Nerche, J., & Harrer, M. (2018). Towards an open source stack to create a unified data source for software analysis and visualization. In *2018 IEEE Working Conference on Software Visualization (VISSOFT)* (pp. 107–111). doi: 10.1109/VISSOFT.2018.00019

Nierstrasz, O., Ducasse, S., & Gundefinedrba, T. (2005, sep). The story of Moose: an agile reengineering environment. *SIGSOFT Softw. Eng. Notes*, *30*(5), 1–10. doi: 10.1145/1095430.1081707

Oliveira, P., Andrade, R. M. C., Barreto, I., Nogueira, T. P., & Morais Bueno, L. (2021). Issue auto-assignment in software projects with machine learning techniques. In *2021 IEEE/ACM 8th International Workshop on Software Engineering Research and Industrial Practice* (pp. 65–72). doi: 10.1109/SER-IP52554.2021.00018

Queiroz, M., Tallon, P., & Coltman, T. (2024, 01). Data value and the search for a Single Source of Truth: What is it and why does it matter? In *Hawaii International Conference on System Sciences* (pp. 6636–6645).

Saeidi, A. M., Hage, J., Khadka, R., & Jansen, S. (2018). A generic framework for model-driven analysis of heterogeneous legacy software systems. In *Proceedings of the 17th Belgium-Netherlands Software Evolution Workshop*. Belgium-Netherlands. (Extended presentation of the 2017 framework with focus on software evolution)

Saha, R. K., Khurshid, S., & Perry, D. E. (2015). Understanding the triaging and fixing processes of long lived bugs. *Information and Software Technology*, *65*, 114–128. Retrieved from https://www.sciencedirect.com/science/article/pii/S0950584915000531 doi: https://doi.org/10.1016/j.infsof.2015.03.002

Sarkar, A., Rigby, P. C., & Bartalos, B. (2019). Improving bug triaging with high confidence predictions at Ericsson. In *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)* (pp. 81–91). doi: 10.1109/ICSME.2019.00018

Siala, H. A., Lano, K., & Alfraihi, H. (2024). Model-driven approaches for reverse engineering—a systematic literature review. *IEEE Access*, *12*, 62558–62580. doi: 10.1109/ACCESS.2024.3394732

Sun, X., Li, B., Li, B., & Wen, W. (2012). A comparative study of static CIA techniques. In *Proceedings of the Fourth Asia-Pacific Symposium on Internetware*. New York, NY, USA: Association for Computing Machinery. doi: 10.1145/2430475.2430498

Wang, H., Tang, Z., Tan, S. H., Wang, J., Liu, Y., Fang, H., ... Wang, Z. (2024). Combining structured static code information and dynamic symbolic traces for software vulnerability prediction. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. New York, NY, USA: Association for Computing Machinery. doi: 10.1145/3597503.3639212

Xia, X., Lo, D., Ding, Y., Al-Kofahi, J. M., Nguyen, T. N., & Wang, X. (2017). Improving automated bug triaging with specialized topic model. *IEEE Transactions on Software Engineering*, *43*, 272–297.

Yau, S.-S., Collofello, J., & MacGregor, T. (1800). Ripple effect analysis of software maintenance. In *Unknown host publication title* (pp. 60–65). IEEE. (COMPSAC '78, IEEE Comput Soc Int Comput Software & Appl Conf, 2nd, Proc ; Conference date: 13-11-1978 Through 16-11-1978)

Zhang, T., Chen, J., Jiang, H., Luo, X., & Xia, X. (2017). Bug report enrichment with application of automated fixer recommendation. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)* (pp. 230–240). doi: 10.1109/ICPC.2017.28

About the authors

Azam Mahdipour is a Master's student at McMaster University (Canada), holding a Bachelor of Science in Software Engineering and a Master's degree in Computer Networks and Security. Her research interests include software development and testing, maintenance, and visualization.You can contact the author at mahdipoa@mcmaster.ca.

Vera Pantelic is a Senior Principal Research Engineer at the McMaster Centre for Software Certification (McSCert) and the McMaster Automotive Resource Centre (MARC), as well as Adjunct Assistant Professor at McMaster University. Her main research interests include the development and certification of safety-critical software systems, model-based development, automotive software, and supervisory control of discrete event systems. You can contact the author at pantelv@mcmaster.ca.

Sébastien Mosser is Associate Professor of Software Engineering at McMaster University (Ontario, Canada). At McMaster, he is Associate Chair of the Department of *Computing and Software* (CAS) and Associate Director of McSCert (*McMaster Centre for Software Certification*). His research interests are related to software engineering, software composition, domain-specific languages and modelling at large. You can contact the author at mossers@mcmaster.ca.