Software Logging and Observability

Bachar RIMA

Equipe MAREL
En collaboration avec
Berger-Levrault

07/11/2023

Table of Contents

Software Logging – Background (1)

- *How can we know what led a program to fail?*
- *How can we know what happened when a system is hacked/breached by an unauthorized party?*
- *How do we know which parts of our program require the most computation power, memory usage, storage capacity, I/O bandwidth, etc.?*
- *How do we know which parts of our system are mostly used and require optimization/scaling?*
- *How do we know which users of website prefer a specific product over another?*
- *Etc.*

First Answer: use **printing statements** with different values depending on the event and its context. We say these printing statements leave **logs** behind them in the console, allowing us to understand what happened.

Software Logging – Background (2)

```java
public boolean addUser(int id, String name) {
    // operation entry trace with provided data
    // event = adding a user to a database
    // context = id and name
    System.out.println("Entered addUser() with id " + id + " and name " + name);

    // creation of user
    User user = new User(id, name);

    // addition of user
    ...

    if (additionResult == true)
        // trace in case of success
        // event = successfully adding a user to a database
        // context = user
        System.out.println("Successfully added user " + user);
    else
        // trace in case of failure
        // event = failed to add a user to a database
        // context = user
        System.out.println("Failed to add user " + user);
}
```

Software Logging – Background (3)

```
public void someOperation() {
    t1 = DateTime.Now();

    // the actual operation implementation
    ...

    t2 = DateTime.Now();

    // execution time trace
    // event = execution success with execution time
    // context = t2 - t1 (the actual time)
    System.out.println("Finished executing someOperation after "
        + (t2 - t1) + " seconds");
}
```

Software Logging – Background (4)

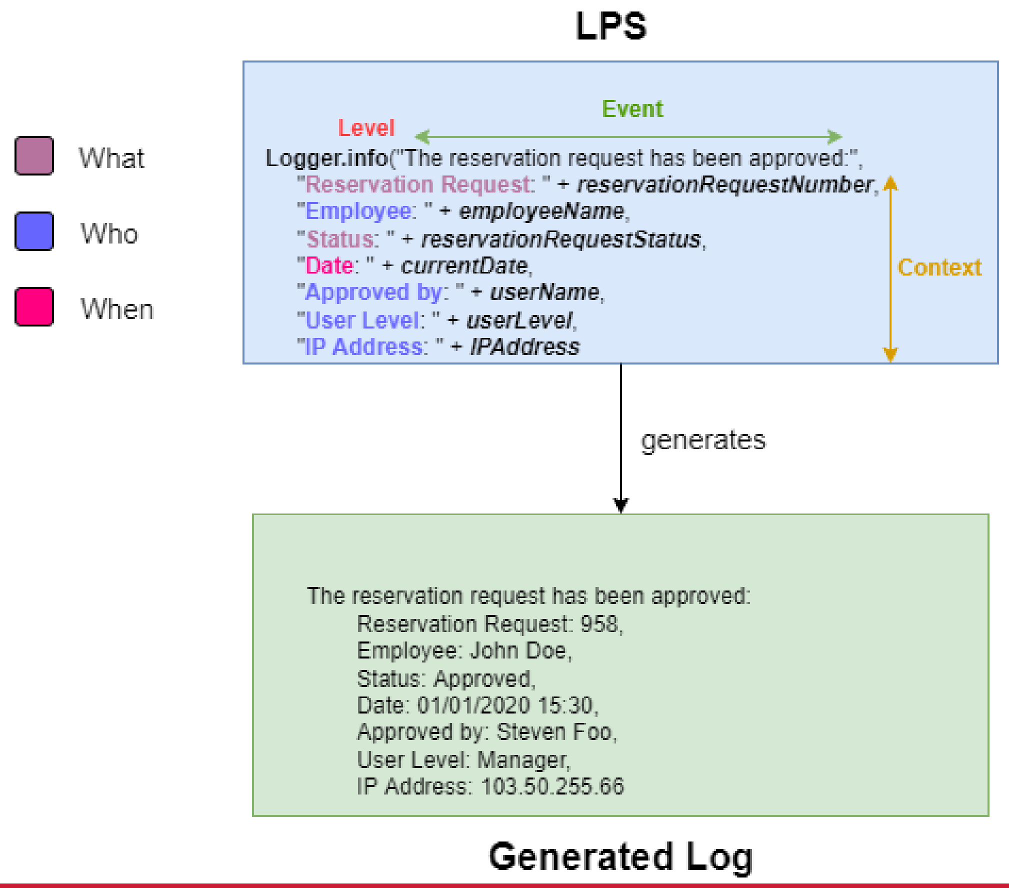
Problems with traditional printing statements

- **Ephemeral**: once the application is done executing, the printed statements are cleared from the console.
- **No verbosity/criticality control**: not all messages need to be printed for all events in all scenarios. For example, in a production environment we only want to see information about the flow of the application (e.g., progress details) while filtering debugging information that are usually available in a development/testing environment.
- **No printing message distribution**: printed statements are only displayed in the standard output/error console. They cannot be dispatched to other destinations simultaneously.

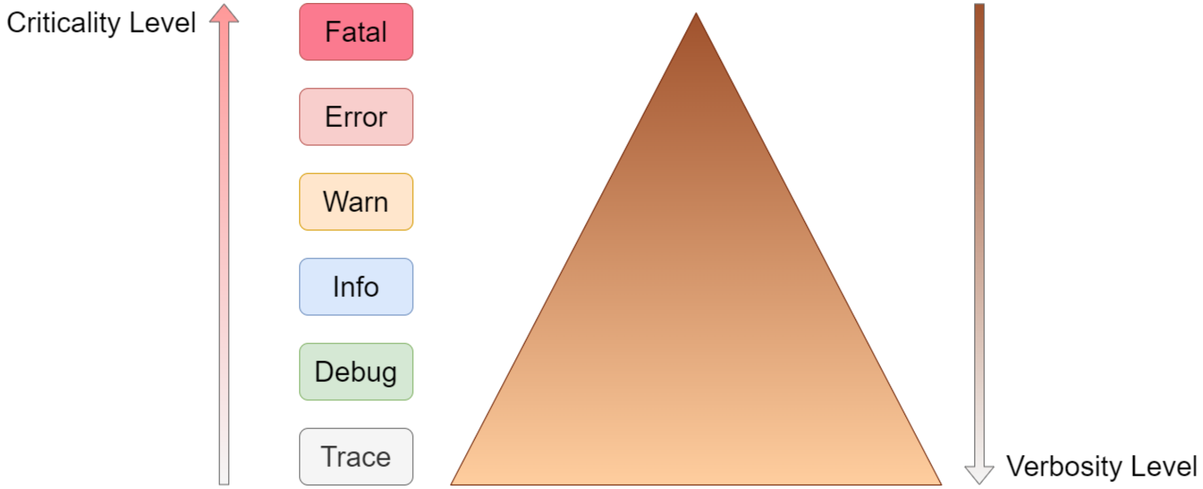
Solution: use logging utilities which can overcome all the above limitations (e.g., java.util.logging, Log4J, SLF4J, etc.) by injecting **Log Printing Statements** (**LPSs**) into a project's code.

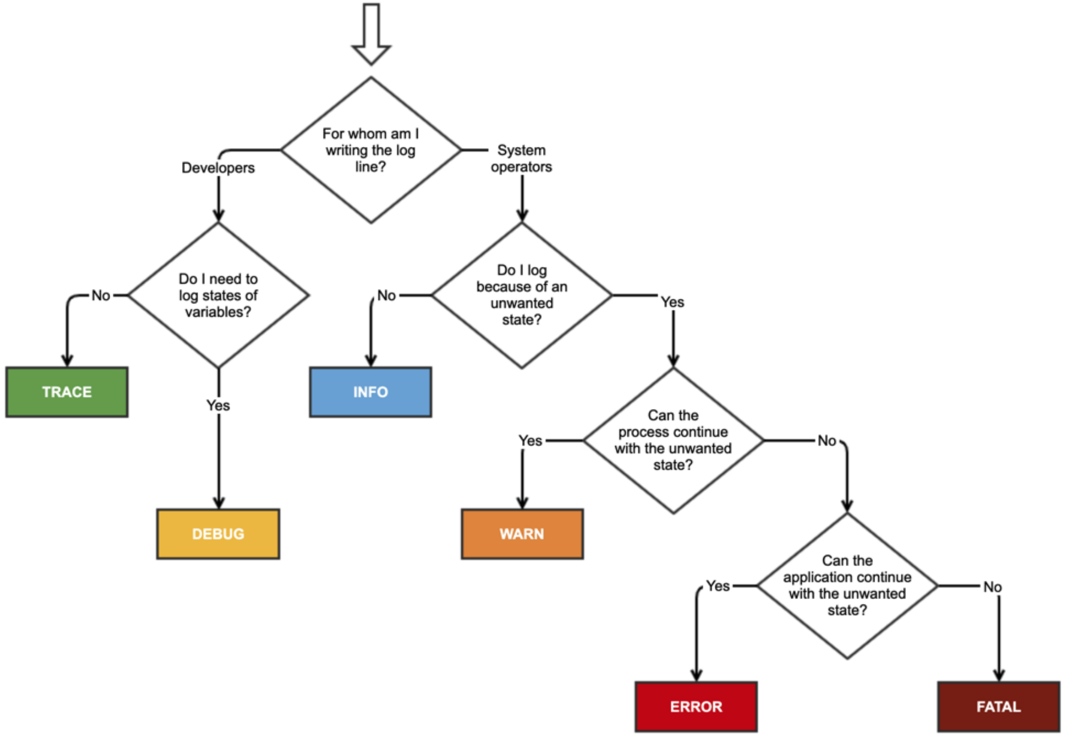
Software Logging – Context: What is logging?

- **Log Printing Statements** (**LPSs**): logging operations injected into a system's source code to extract information related to the fulfillment of a **logging goal** and generating **logs** that would be further analyzed to react accordingly.
- **LPS event**: the event captured by an LPS.
- **LPS context**: the actual programming elements (variables, method calls, etc.) used to build the LPS event's context, generally consisting of five parts:
  1. ***Who***: who triggered the event
  2. ***When***: when was the event triggered
  3. ***Where***: where the event was trigged (e.g., GUI widget)
  4. ***What***: what event information to retrieve (i.e., event type and other event-related data)
  5. ***Why***: why was the event triggered.
- **LPS level**: the criticality of the generated log, controlling what kind of information will be included in each part of the LPS event.



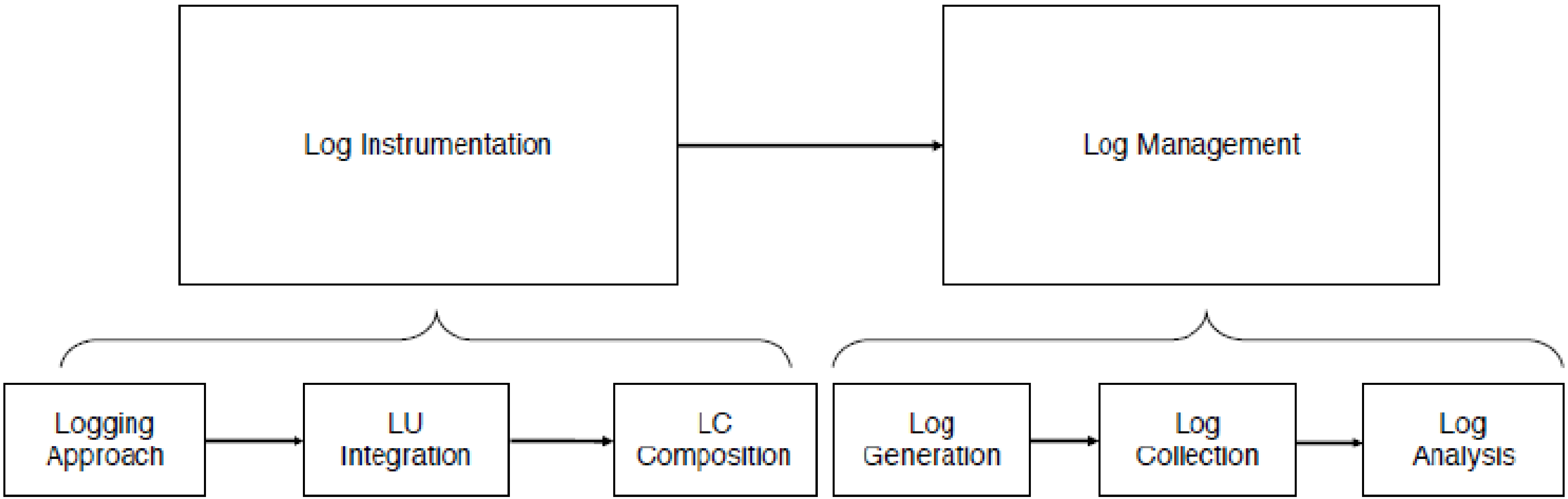
Software Logging – Context: LPS Criticality Level





https://stackoverflow.com/questions/2031163/when-to-use-the-different-log-levels

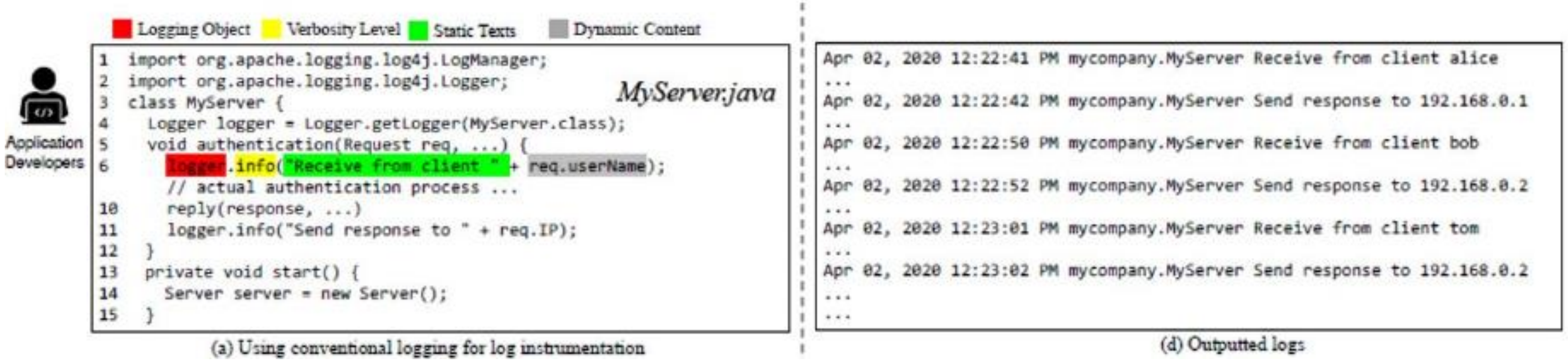
Software Logging – Context: The Logging Process



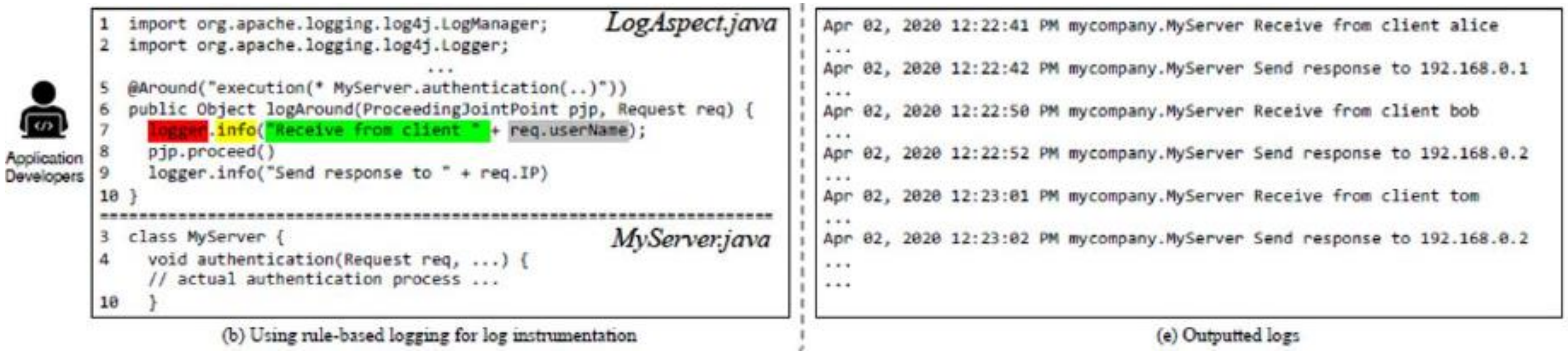
Chen, B. (2020). Improving the Logging Practices in DevOps.

Software Logging – Context: Logging Approaches (1)

- **Conventional Logging**: free-form logging with the logging code, scattered across the codebase and tangled with the feature code.



- **Rule-based Logging**: free-form logging with the logging code, modularized into separate files (e.g., Aspect-Oriented logging).



Chen, B. (2020). Improving the Logging Practices in DevOps.

Software Logging – Context: Logging Approaches (2)

- **Distributed Tracing**: structured logging using end-to-end traces to capture the full picture across multiple machines and processes in a distributed system (e.g., OpenTracing API, Dapper by Google, …)

Chen, B. (2020). Improving the Logging Practices in DevOps.

Software Logging – Context: Logging Approaches (3)

Dimension/Logging Approach	Conventional Logging	Rule-based Logging	Distributed Tracing
Who	SUS developers	SUS developers	Logging Library developpers
Filtering	Verbosity Level	Verbosity Level	Sampling
Format	Free Form	Free Form	Structured
Domain	General	General	Distributed Systems
Flexibility	High	Low	Medium
Scattering	High	Low	Low

Chen, B. (2020). Improving the Logging Practices in DevOps.

Software Logging – Context: Logging Approaches (4)

- **Problems**:
    1. *“The majority of software project still adopt conventional logging approaches”* (*Chen & Jiang (2020), Studying the Use of Java Logging Utilities in the Wild*).
    2. *“Steep learning curve of the other approaches”* (*Chen, B. (2020). Improving the Logging Practices in DevOps*).

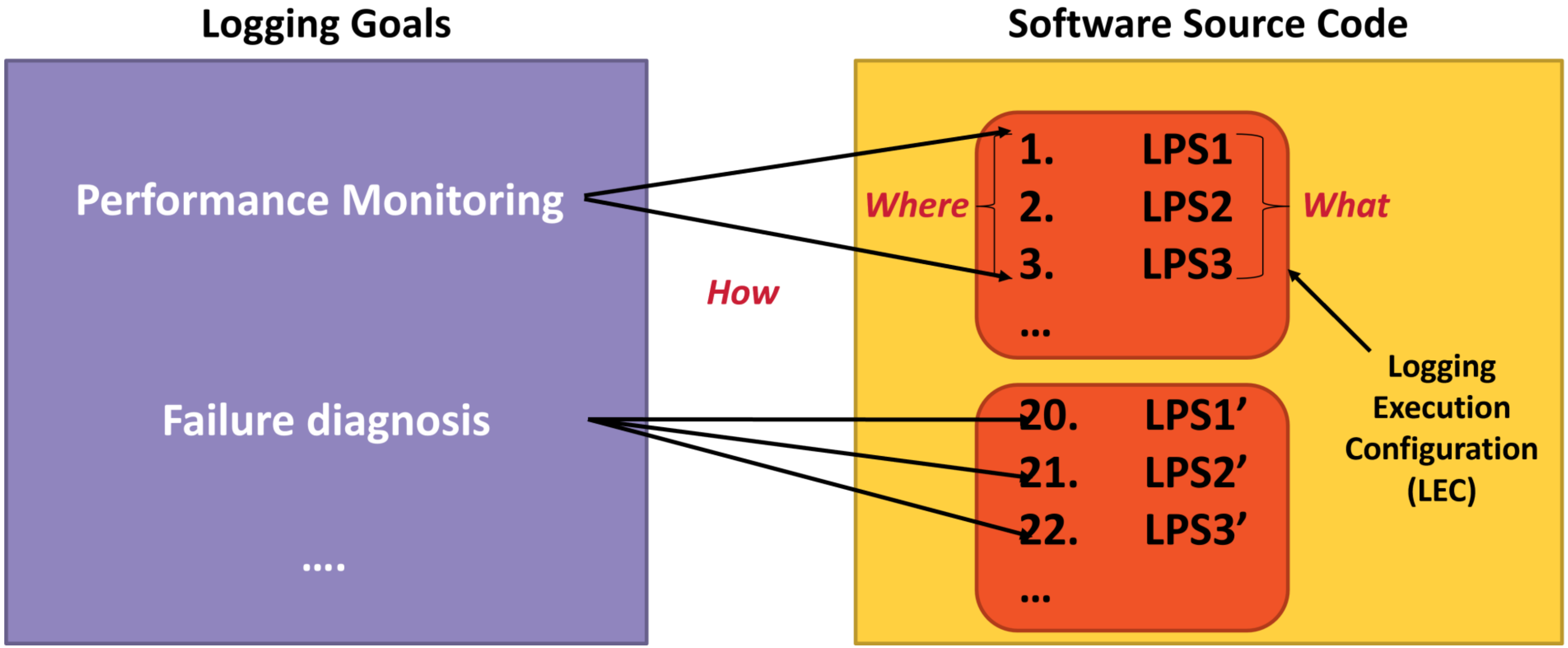
- **Active research question**: *“How to suggest the appropriate logging approach(es) for a* ***System under Study*** *(**SUS**)’s specific logging scenarios and logging goals?”*

Software Logging – Context: Logging Utilities Integration

- **Evolution over the years**: from simple *printf()* statements to more complex and commonly-used libraries:
  - **Java**: SLF4J, Log4j, …
  - **C++**: Spdlog, …
- **New features**: persistence, thread-safety, verbosity levels, formatting, …
- **Different LU(s) provide different functionalities** → developers may integrate one/many existing general-purpose or specialized LU(s), or even develop their own depending on the usage context.
- **Each project may contain one or more LU(s), each providing many different configuration options** → developers may need to properly configure the LU(s) to gather enough data to:
  1. Gain full visibility of the SUS
  2. Minimize the performance overhead and storage requirements.
- **Active research questions**:
  1. *"Which LU(s) should be used, given a SUS?"*
  2. *"How to migrate to other/newer LU(s), given a SUS?"*
  3. *"How to automate the configuration management across multiple LU(s), given a SUS?"*

Software Logging – Context: Logging Code Composition

- **Where to Log**: determine the appropriate logging points in the source code to inject LPSs.
- **What to Log**: determine how to build the LPSs (event, context, criticality)
- **How to Log**: choose a logging strategy to inject the LPSs into their appropriate logging points.

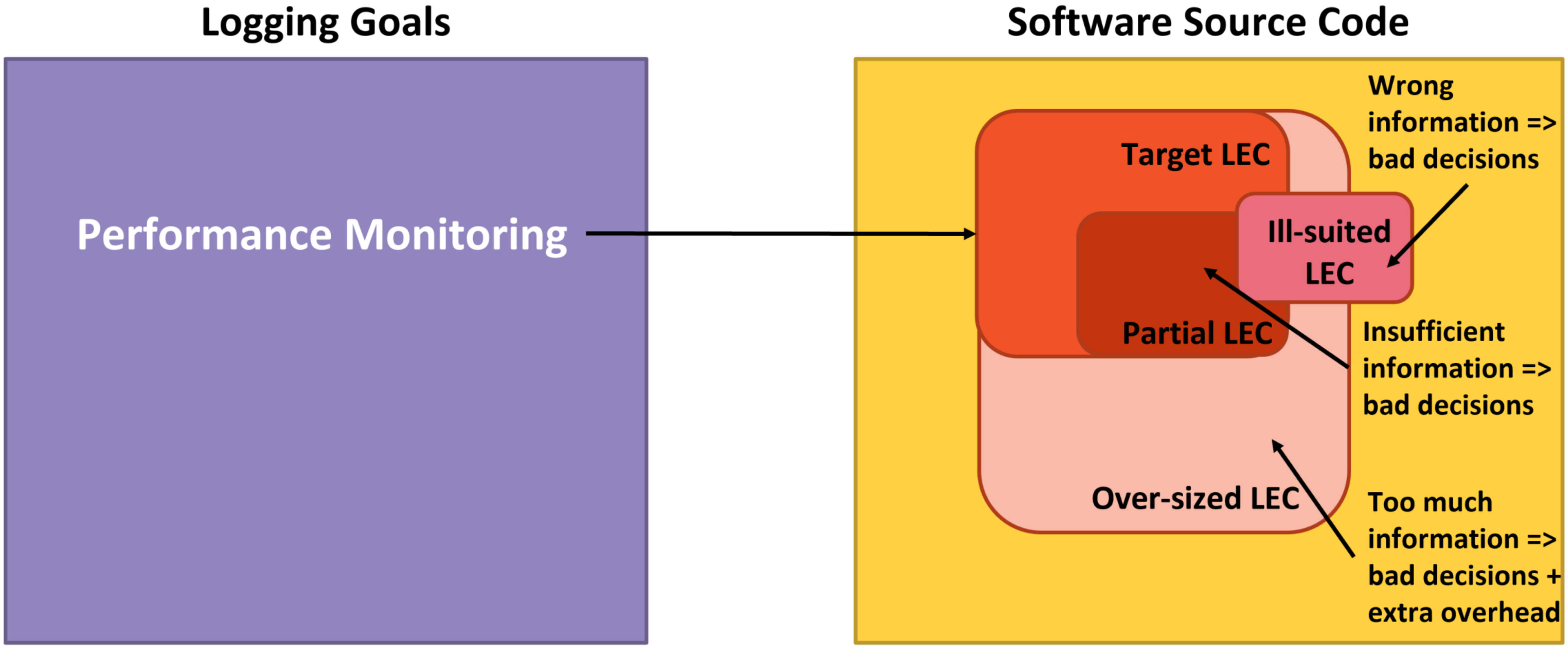


Software Logging – Problem Statement: Guidelines & Automation

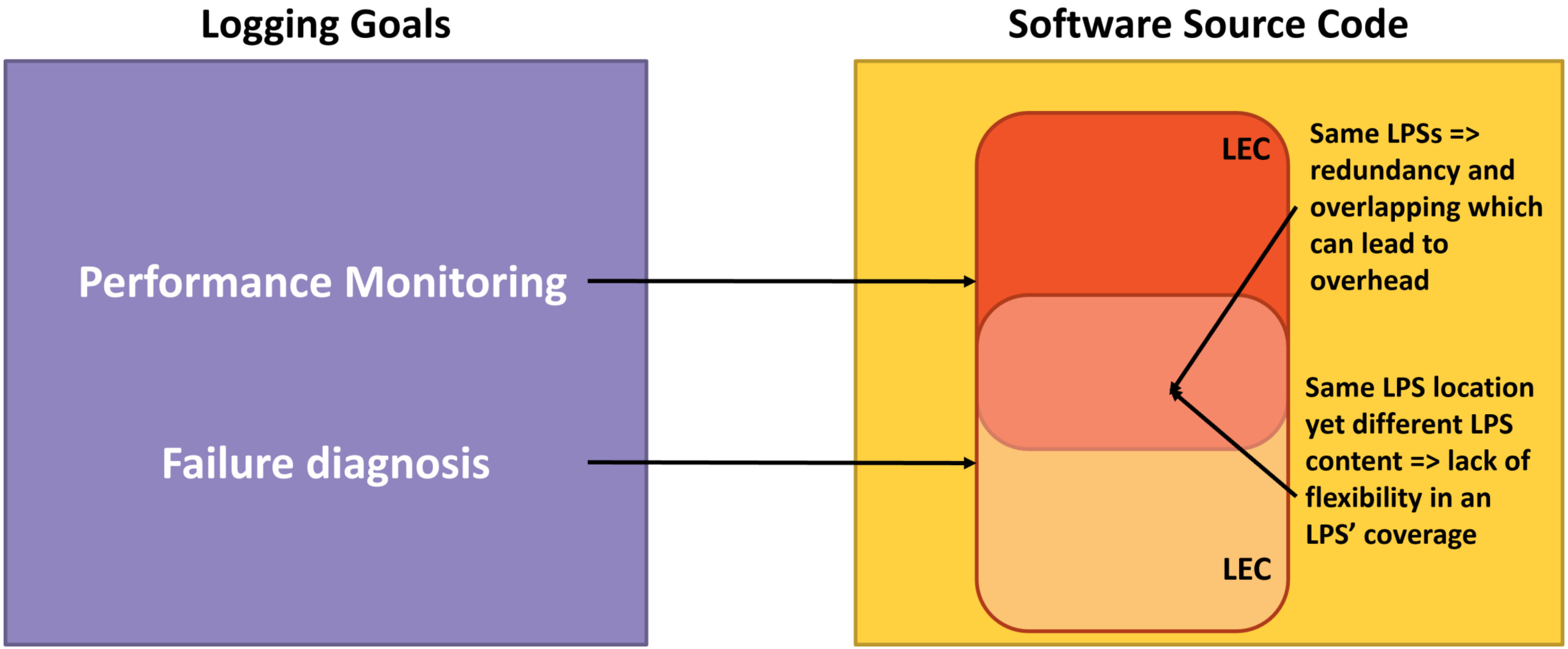
- **Very few systematic guidelines on log instrumentation exist**, even fewer are applied in practice.

- **Automated approaches for log instrumentation exist**, but they are mostly **partial** or **costly**:
  1. Handling one logging goal at a time.
  2. Don't necessarily address all aspects of logging code composition:
     - ❑ Where to log only;
     - ❑ What to log only;
     - ❑ What log levels to use only;
     - ❑ Any incomplete combination of the above.

Software Logging – Problem Statement: Coverage Mismatch between Logging Goals and LECs



Software Logging – Problem Statement: Flexibility, Overlapping, and Redundancy of LPSs



SoftScanner – Motivation

- **Very few systematic guidelines on log instrumentation exist**, even fewer are applied in practice → need for more automation to reduce developer effort and error

- **Automated approaches for log instrumentation exist**, but they are mostly **partial** or **costly**:
  1. Handling one logging goal at a time → need for logging support for multiple logging goals at once
  2. Need for an approach covering all aspects of LPS composition (where, what, how, and at which level)

- **Non-optimal coverage for a logging goal by an LEC** may result in bad decisions, and possibly extra overhead → need for optimization mechanisms to generate LECs that optimally cover their logging goals

- **Redundant and overlapping LPSs** may lead to overhead and lack of flexibility → need for optimization mechanisms to handle redundant and overlapping LPSs

SoftScanner – Overview

- SoftScanner is a log-centric, software analysis and engineering ecosystem
- SoftScanner is semi-automatic → need for more automation to reduce developer effort and error
- Softscanner supports a multi-logging-goal-based log instrumentation approach → need for logging support for multiple logging goals at once
- SoftScanner provides different logging strategies to build LECs for specific logging goals, with different criticality level output filters, while establishing a bi-directional link between the LPSs and their corresponding logs → need for an approach covering all aspects of LPS composition (where, what, how, and at which level)
- SoftScanner provides an LPS optimizer component to deal with redundancy, overlapping, covering mismatch scenarios, and other LPS-related optimization issues →
  - ❑ need for optimization mechanisms to generate LECs that optimally cover their logging goals
  - ❑ need for optimization mechanisms to handle redundant and overlapping LPSs

SoftScanner – Roadmap (1)

Axis 1: logging goals & LECs	Axis 2: Static optimization of LECs
• Make a feature model of all possible logging goals • Select logging goals of interest • Implement at least one logging strategy for each logging goal of interest	• **Goal**: Perform static generation of optimal LECs for multiple logging goals at once • **Approach**: Implement different optimization techniques that balance the trade-off between reducing LEC costs and maximizing its logging goal’s degree of fulfillment

SoftScanner – Roadmap (2)

Axis 3: Dynamic optimization of LECs

- **Goal**: Allow a dynamic generation of LECs where data collection points can be enabled, disabled, and/or defined on the run to fulfill a given set of tracing goals
- **Approach**: Apply information dynamic analysis and coding techniques that allow the dynamic enabling, disabling, and definition of LPSs

Axis 4: Model-Driven-Engineering-based generation of LECs

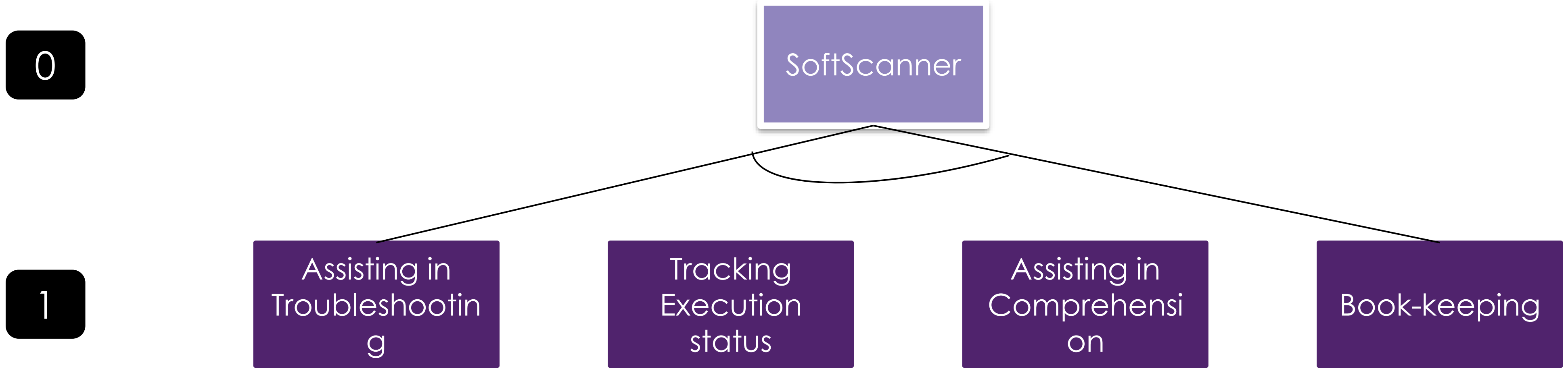
- **Goal**: Make the approach platform-independent by generalizing concepts tackled in the previous axes and making them model-driven using MDE
- **Approach**: Define and reuse a set of metamodels covering the relevant domains (source code, LECs, LPSs, logging goals, …) as well as the necessary mapping rules between them

SoftScanner – Roadmap (3)

Axis 5: Interactive visual platform for LEC adaptation

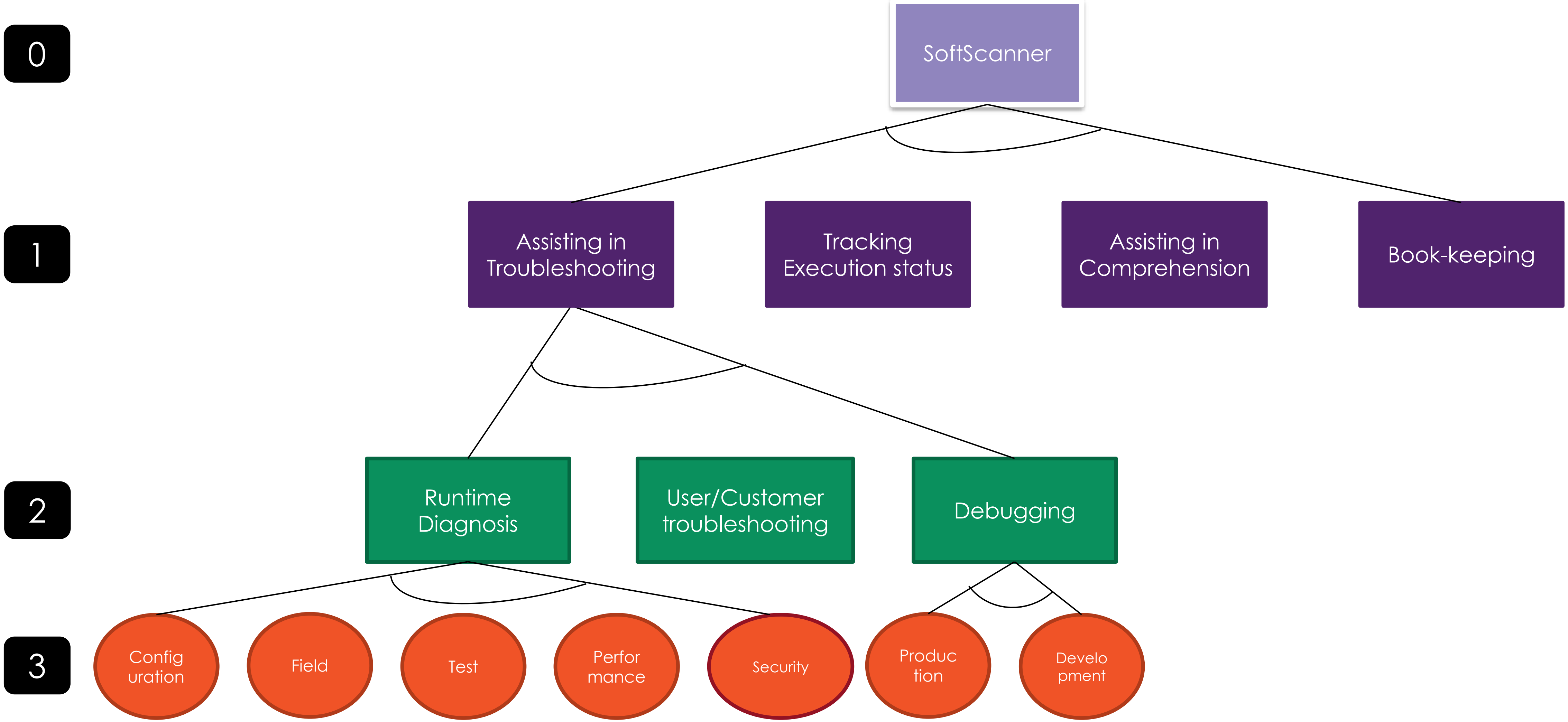
- **Goal**: Allow operators to directly adapt LECs with respect to the desired logging goals.
- **Approach**: Examine existing interactive visualization platforms (e.g., Kibana dashboard) and identify their limitations, then propose a solution to palliate them and integrate it to the project

SoftScanner – Feature Model (1)

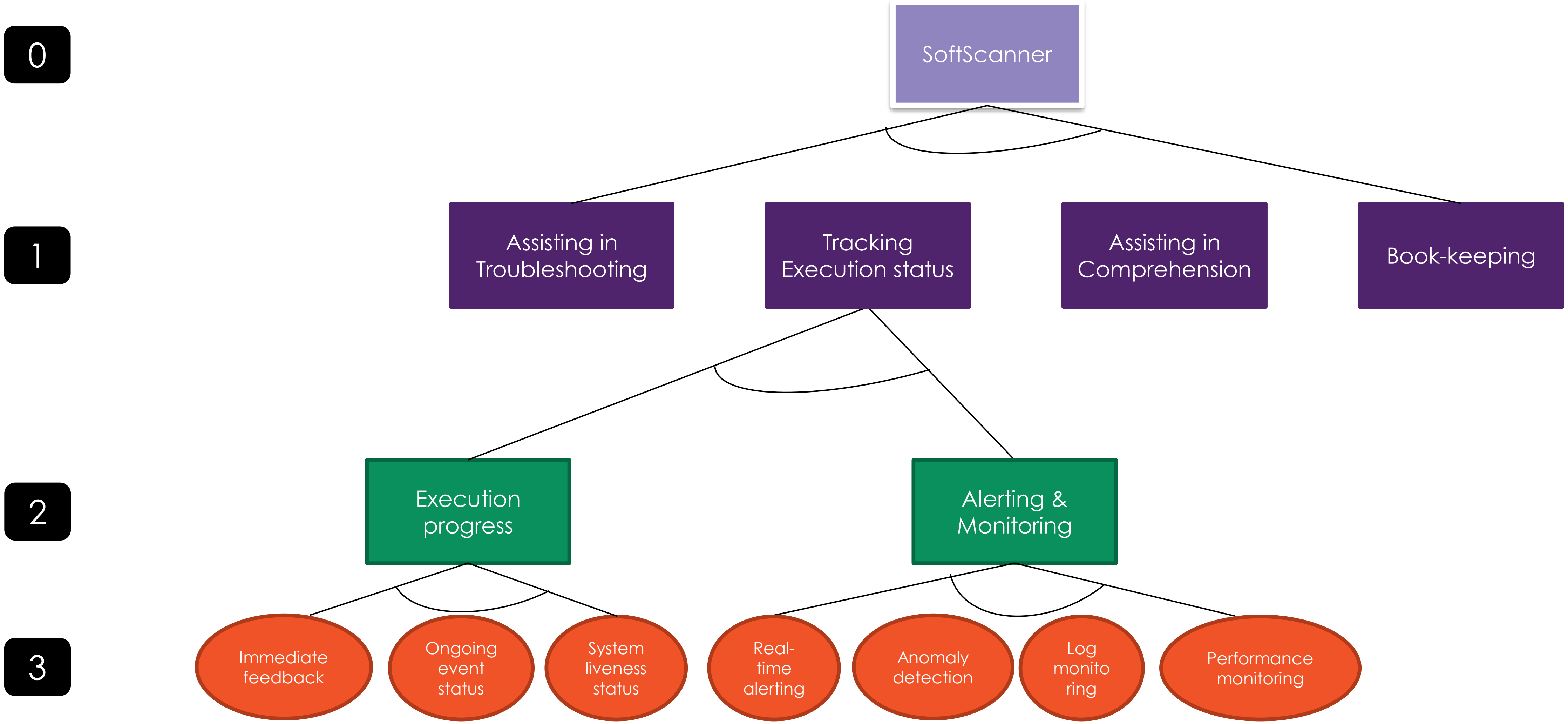


Based on *H. Li et al. (2020). A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives.*

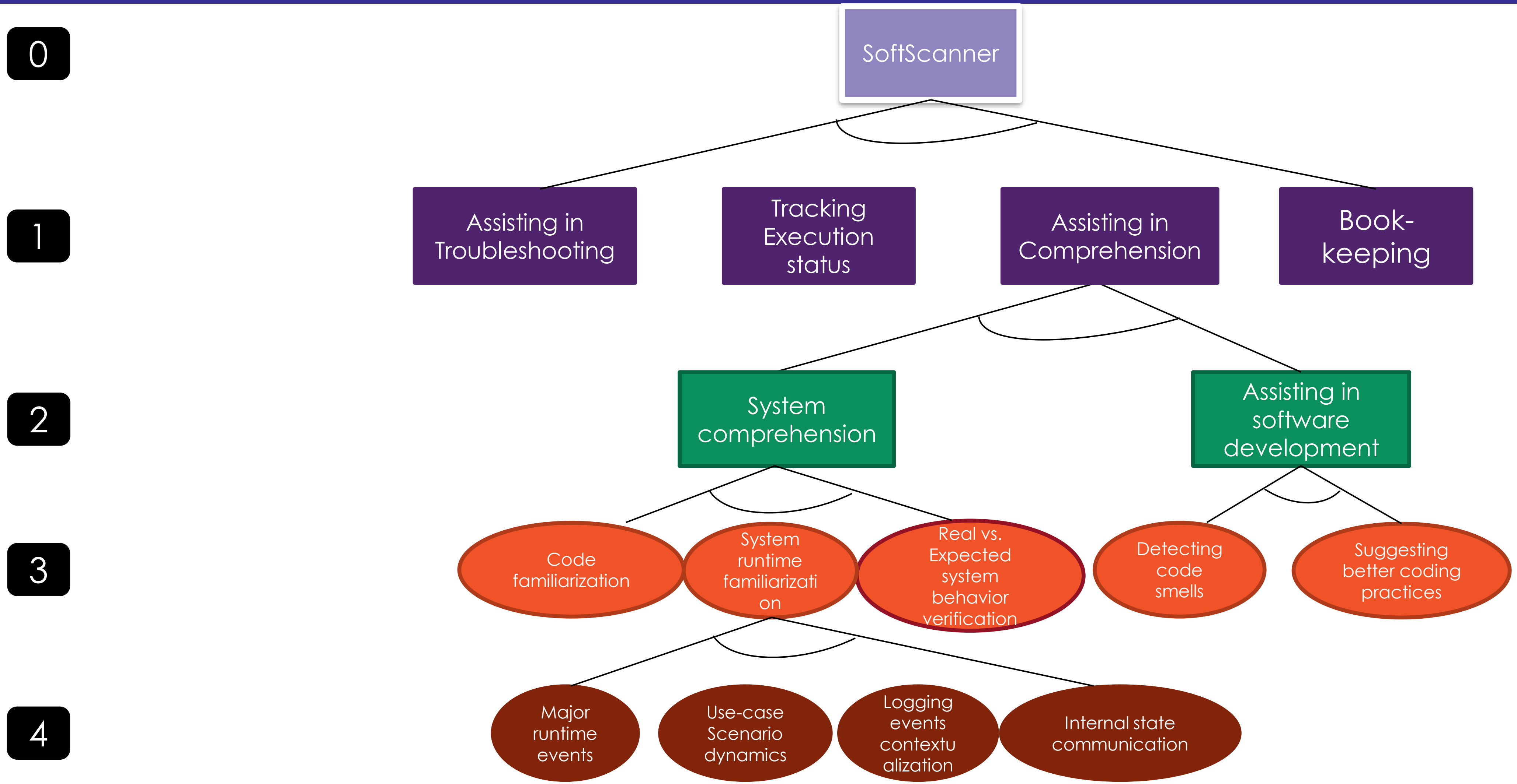
SoftScanner – Feature Model (2)



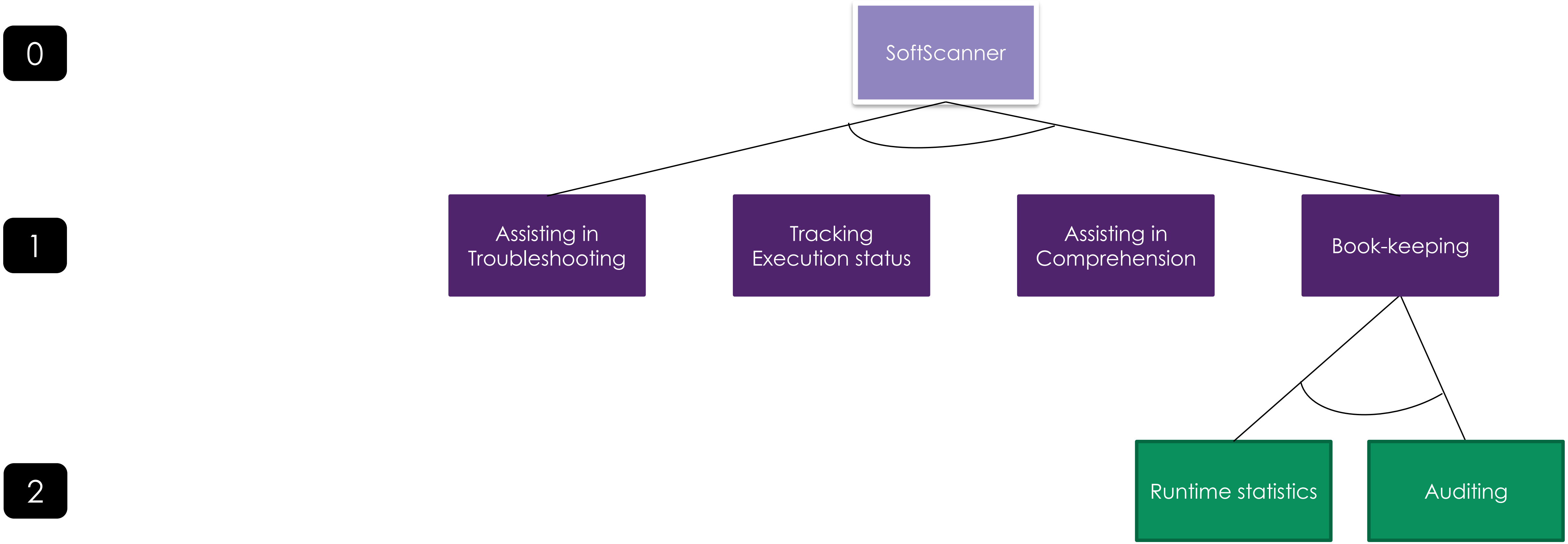
SoftScanner – Feature Model (3)



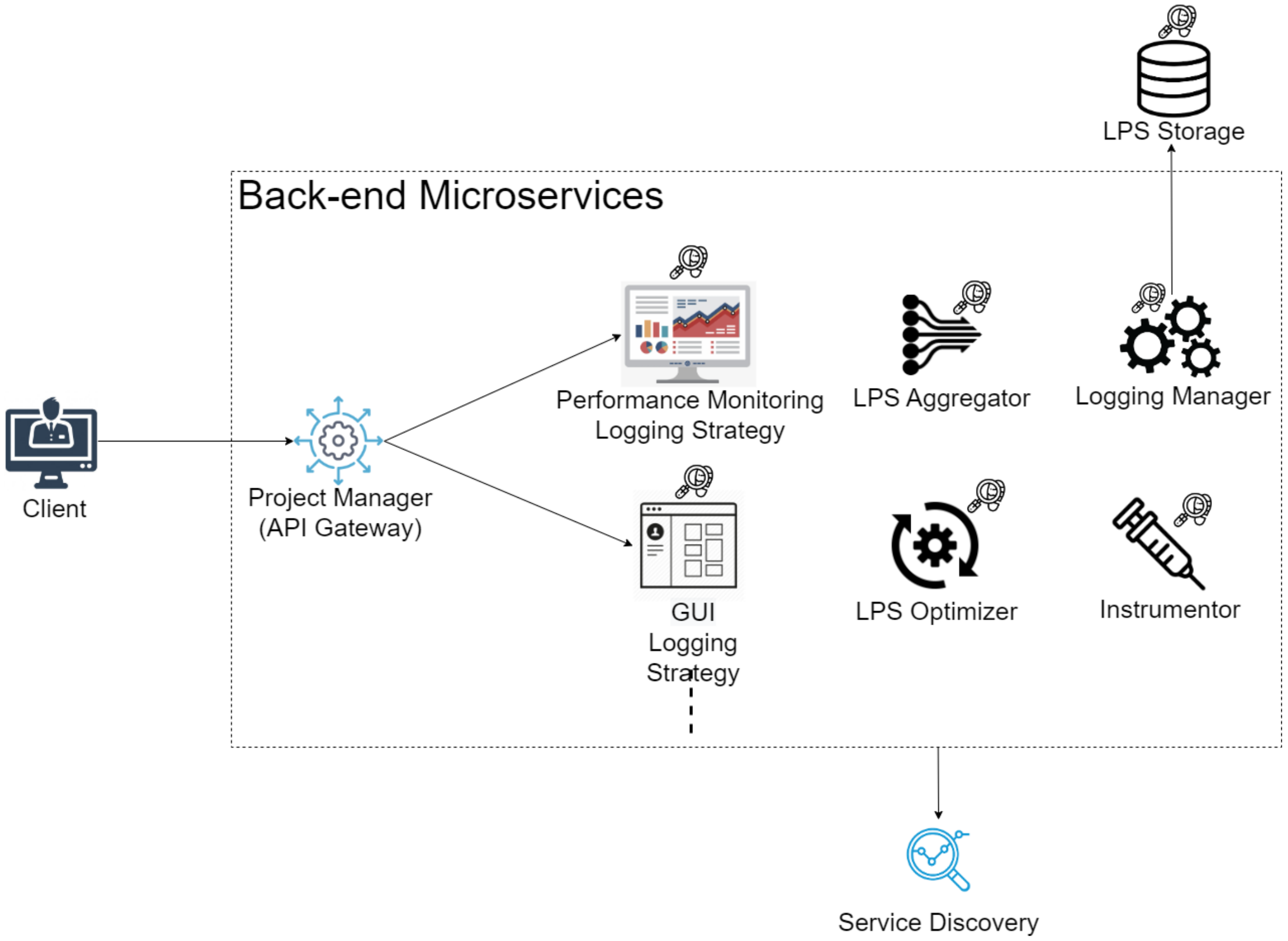
SoftScanner – Feature Model (4)



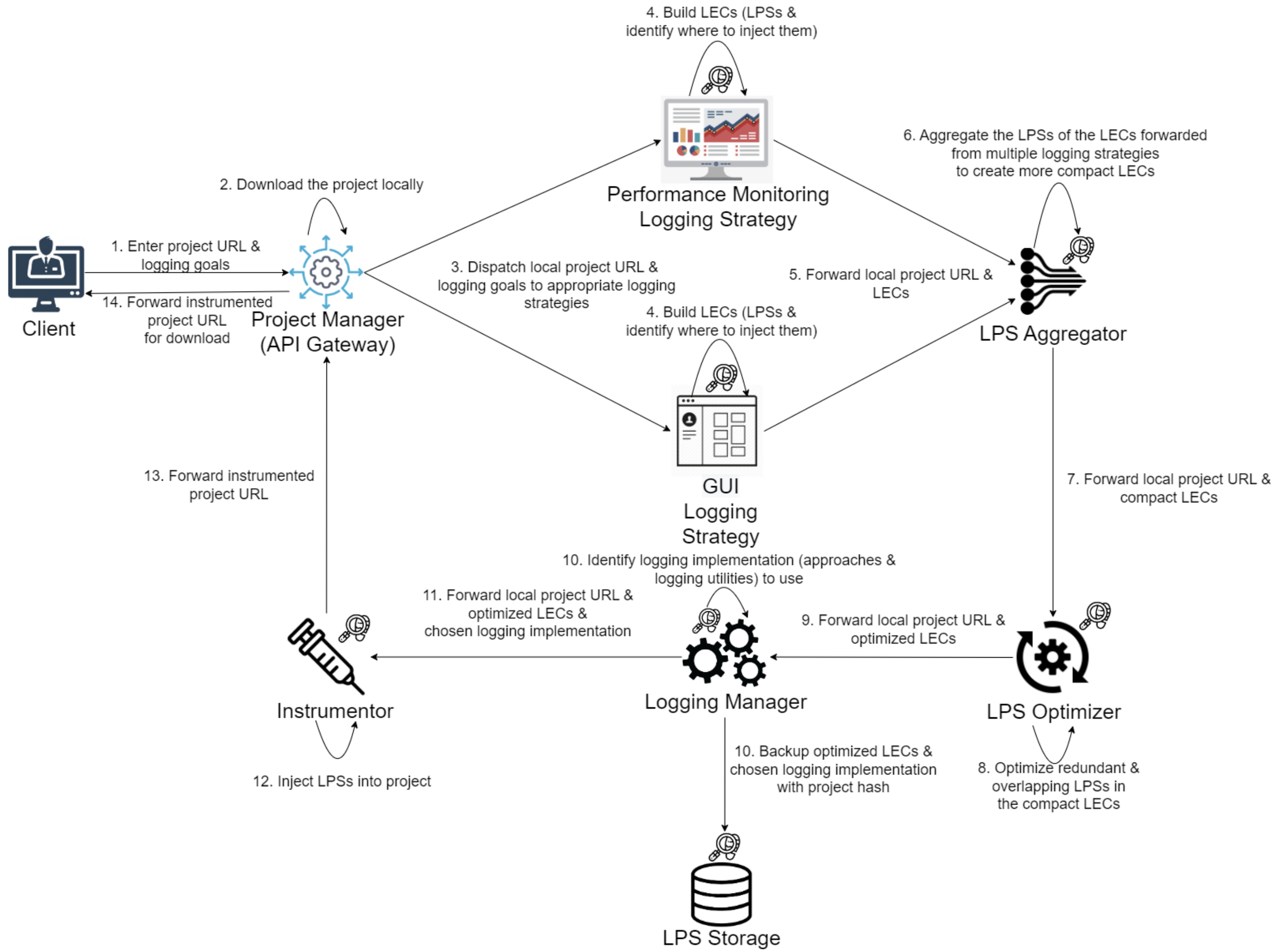
SoftScanner – Feature Model (5)



SoftScanner – Logging Instrumentation Architecture



SoftScanner – Logging Instrumentation Workflow

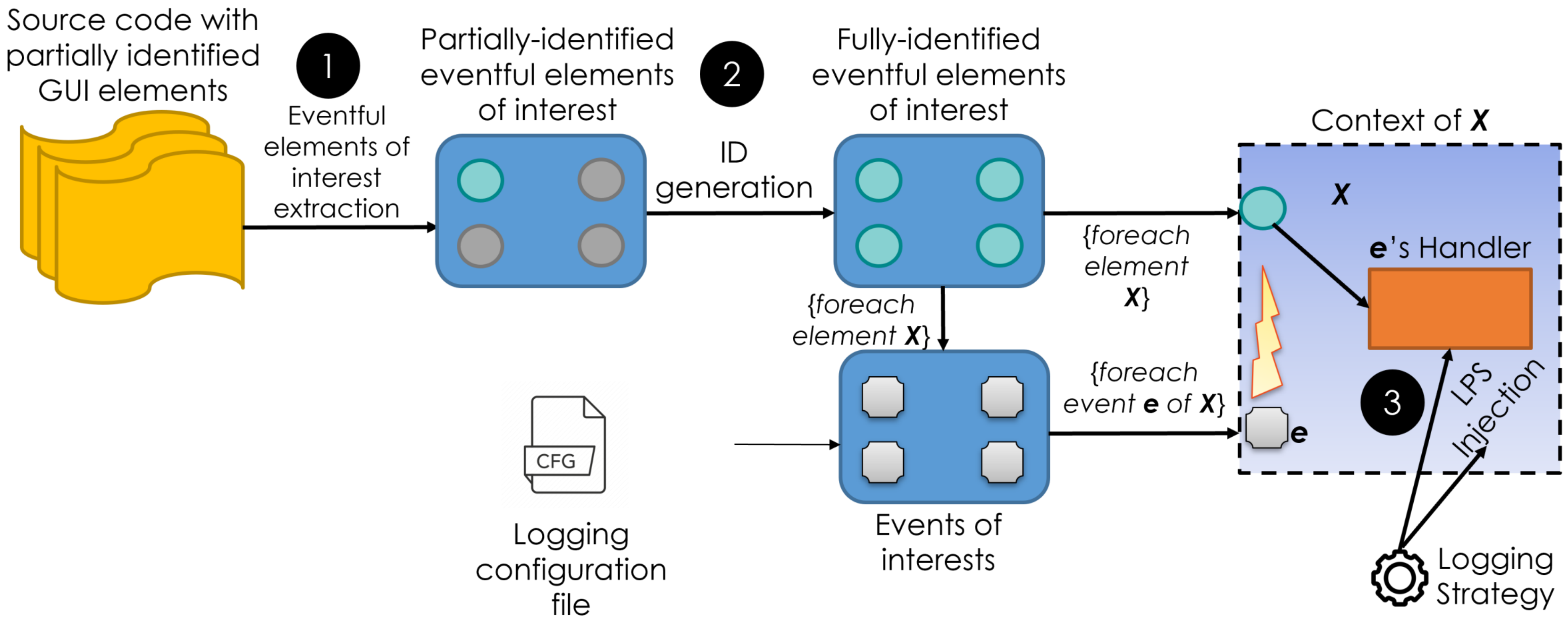


SoftScanner – Event-based GUI Widgets Logging (1)

- **Goal**: provide a logging mechanism for GUI widgets of Angular projects.
- **Methodology**: given a set of target events, automatically log all widgets upon which these events can be triggered.

- **Currently supported events**:
  - **Click**: clickable widgets
  - **Submit**: form submissions (*ngSubmit* for Angular)
  - **Href**: hyper-linkable widgets (+ *routerLink* for Angular)
  - **FocusOut**: meaningfully focusable widgets (e.g., **<input>** widgets like text fields, …, **<textarea>** widgets, …)
  - **Change**: meaningfully changeable widgets (e.g., **<select>** drop-down lists, **<input>** widgets like checkboxes, radio buttons, …)
  - **File**: file uploads

SoftScanner – Event-based GUI Widgets Logging (2)



SoftScanner – Event-based GUI Widgets Logging (3)

Software Observability – Limitations of Software Logging

Overwhelming Volume and Verbosity

- Massive volumes of data make it difficult to store and manage logs efficiently
- High verbosity can obscure important information, making it hard to extract actionable insights

Impact on Performance and Costs

- Logging at a granular level can incur performance overhead
- Storing and processing large volumes of logs can lead to increased storage and processing costs

Real-time Analysis Limitations

- Logs are not always suitable for real-time analysis due to processing delays
- It is challenging to set up real-time alerts based on log data without a significant investment in tooling

Logs in Isolation Lack Context

- Logs often provide limited context when viewed in isolation
- Correlating logs across services in a distributed system can be challenging without proper tools

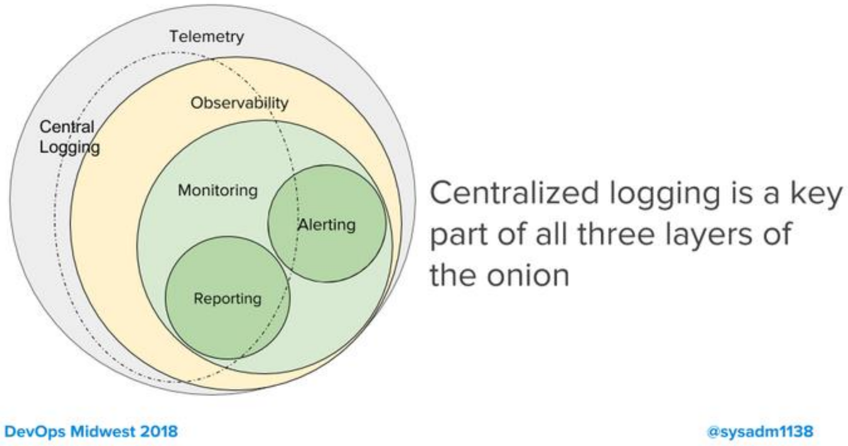
Logs Fall Short on User Journey Insights

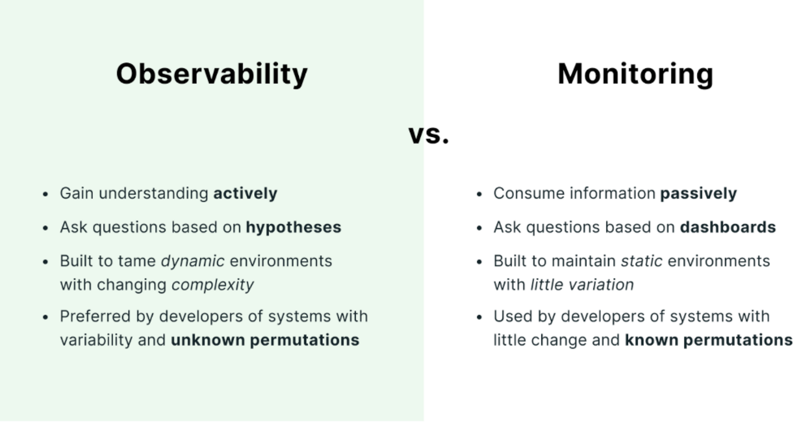
- Logs typically don't capture the end-to-end user journey or transaction flow
- Understanding user behavior or transaction performance is difficult with logs alone

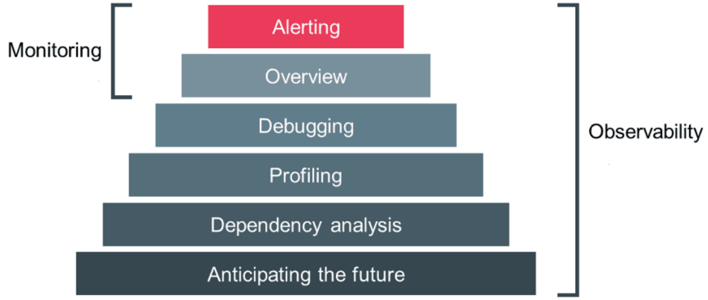
Troubleshooting Complexities

- Sifting through logs to identify issues is time-consuming and often requires expert knowledge
- Without structured logs, troubleshooting can become a needle-in-a-haystack scenario

Software Observability – Introduction







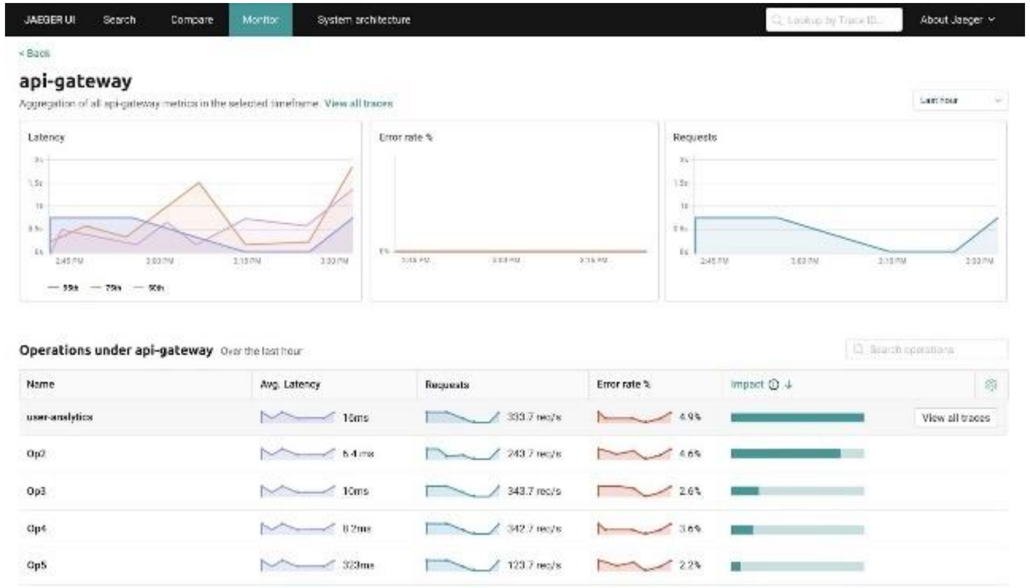
Observability is concerned with understanding what's going on **internally** in a system based on its **outputs**.

Observability implies implementing a **MELT** strategy

Telemetry is the collection of measurements or other data at remote points and their automatic transmission to receiving equipment for further processing.

Telemetry data includes **MELT** and extends to all other data created by a system.

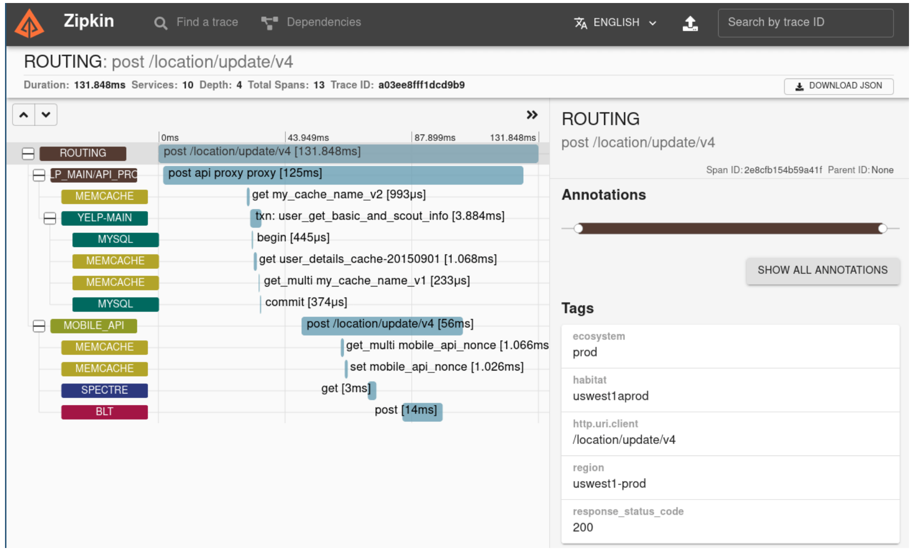
Software Observability – Metrics

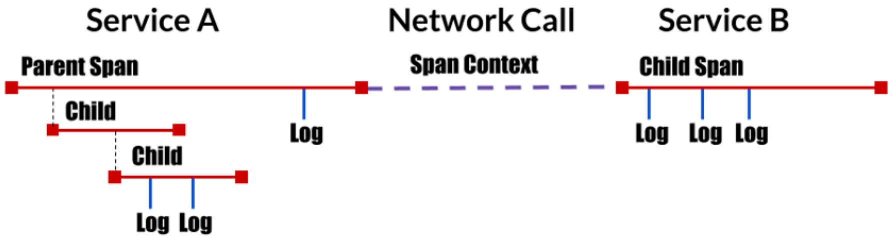


KPIs	Metrics
• All KPIs are Metrics	• All Metrics are not KPIs
• KPIs give a holistic view of the performance of different functions in your organization	• Metrics give you a picture of how different individual activities rolled out within the functions are progressing
• KPIs tell you where exactly your teams stand with respect to the overall business goals	• Individual Metrics do not give any insights on their own
• Examples: Pre-sales KPIs, Email Marketing KPIs, Customer Success KPIs	• Examples: Open Rate, Conversations in the last 2 weeks, Deals lost last quarter

Metrics are measurements collected at regular intervals. It can be error rate, response time, CPU mean usage, etc.

Software Observability – Traces

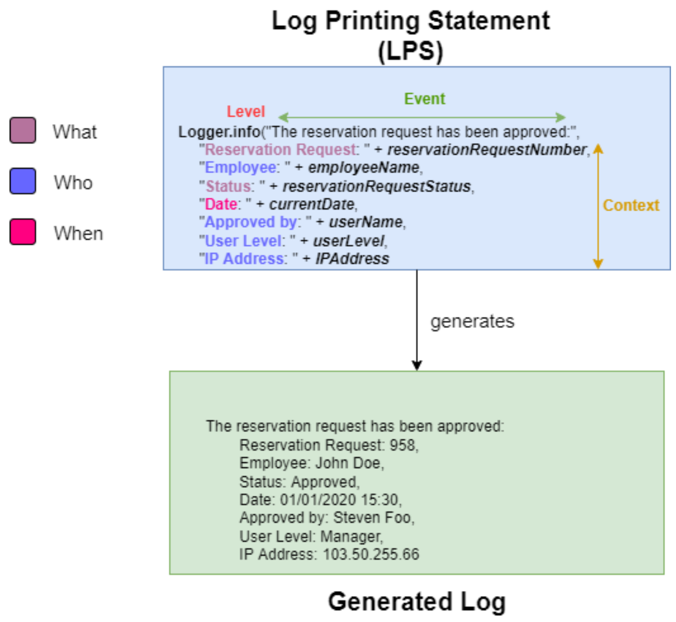


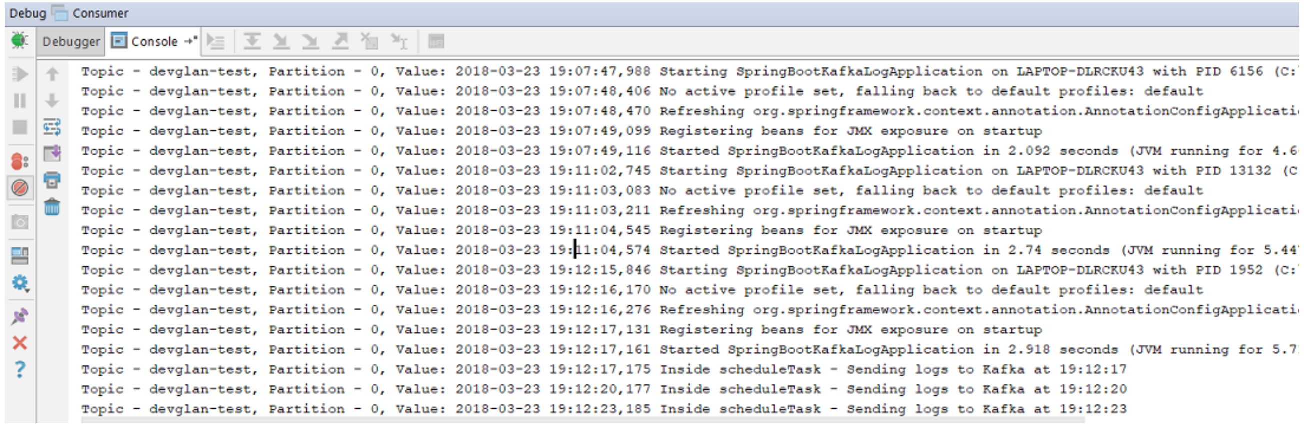


Traces are a collection of **spans** covering operations occurring during the completion of an **end-to-end request** to reconstitute what happened. Traces can be potentially augmented with **logs**.

A **span** is a discrete unit of work that is tracked within a **trace**. A trace, therefore, is a series of causally related and potentially nested spans covering an end-to-end request within a system.

Software Observability – Logs

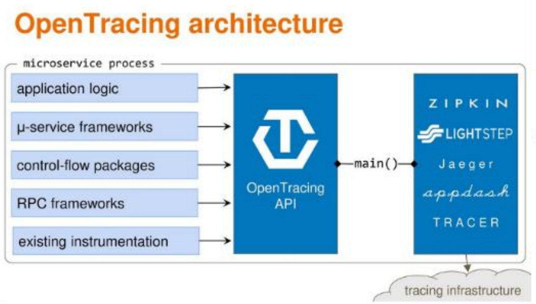


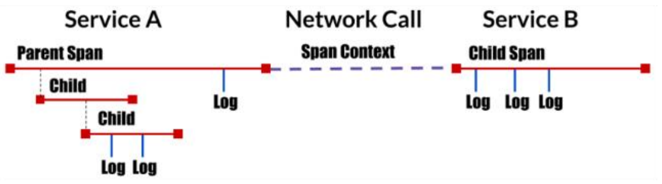


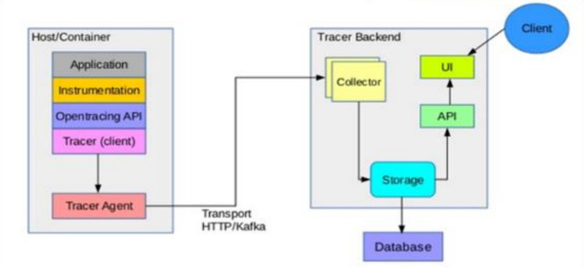
Logs are a collection of events generated by **Log Printing Statements** (**LPS**) into the code. An LPS can be added manually or automatically instrumented into the code.

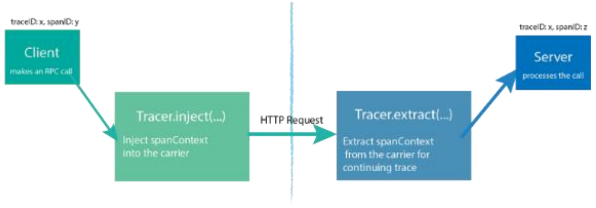
Logs provide **micro-level** details and context about a **specific span** while traces provide **macro-level** details and context across **multiple spans**

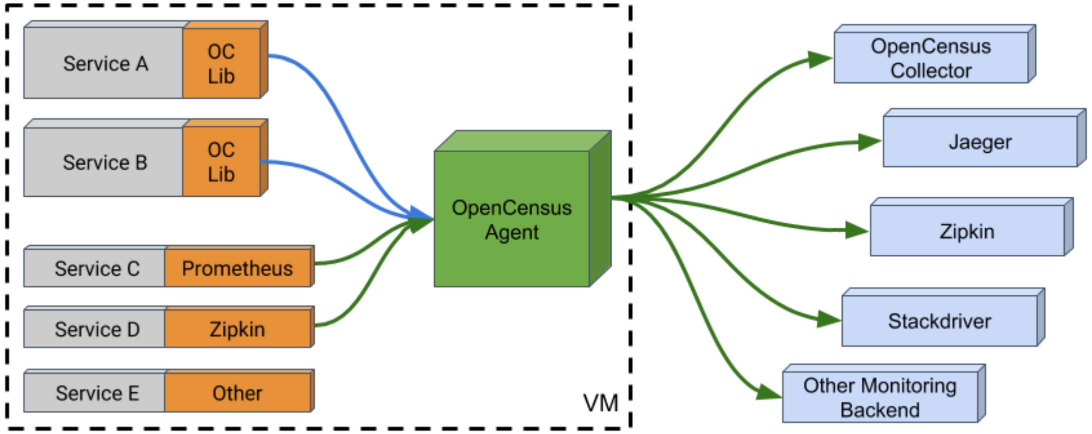
Software Observability – Industry Standards: OpenTracing and OpenCensus (*In the past*)



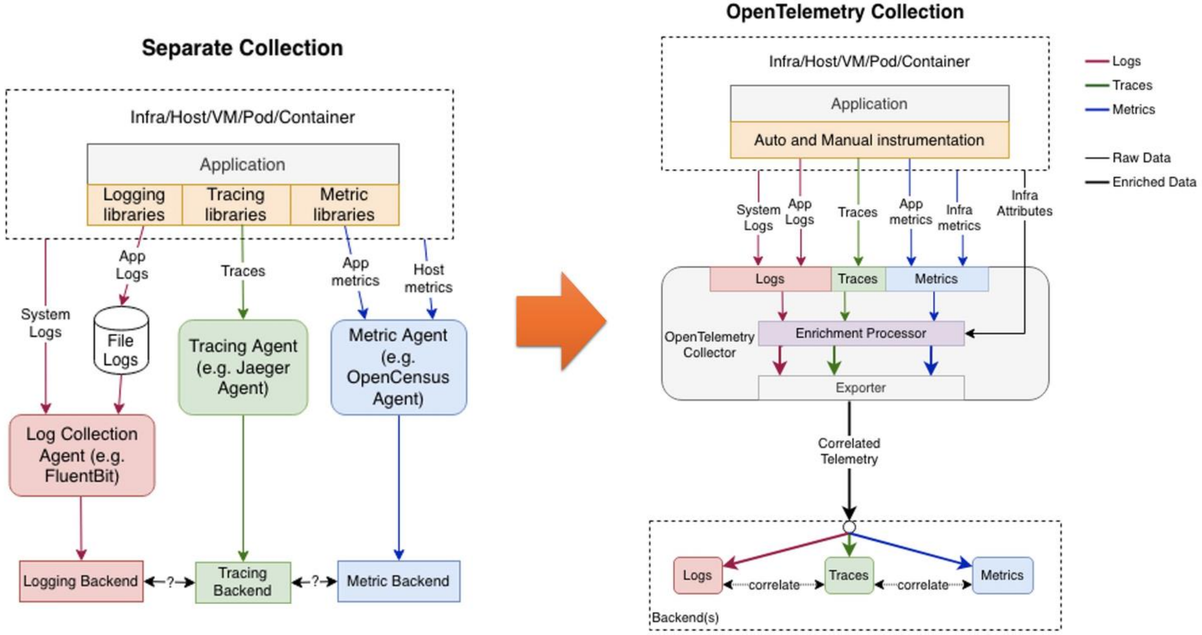


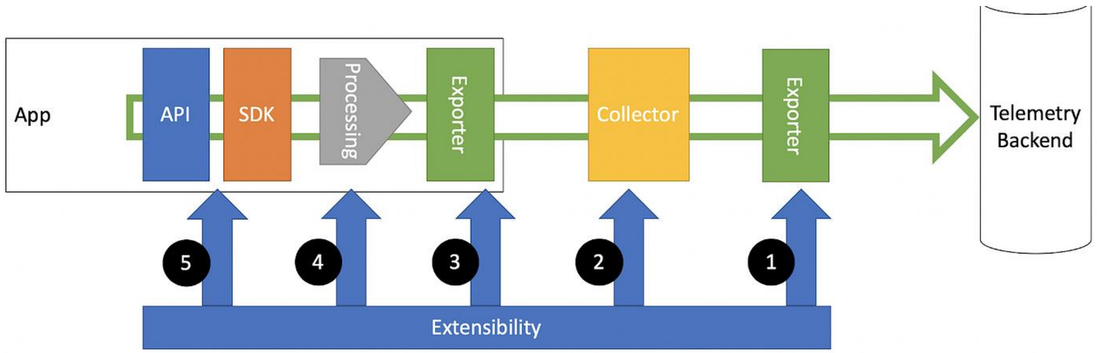


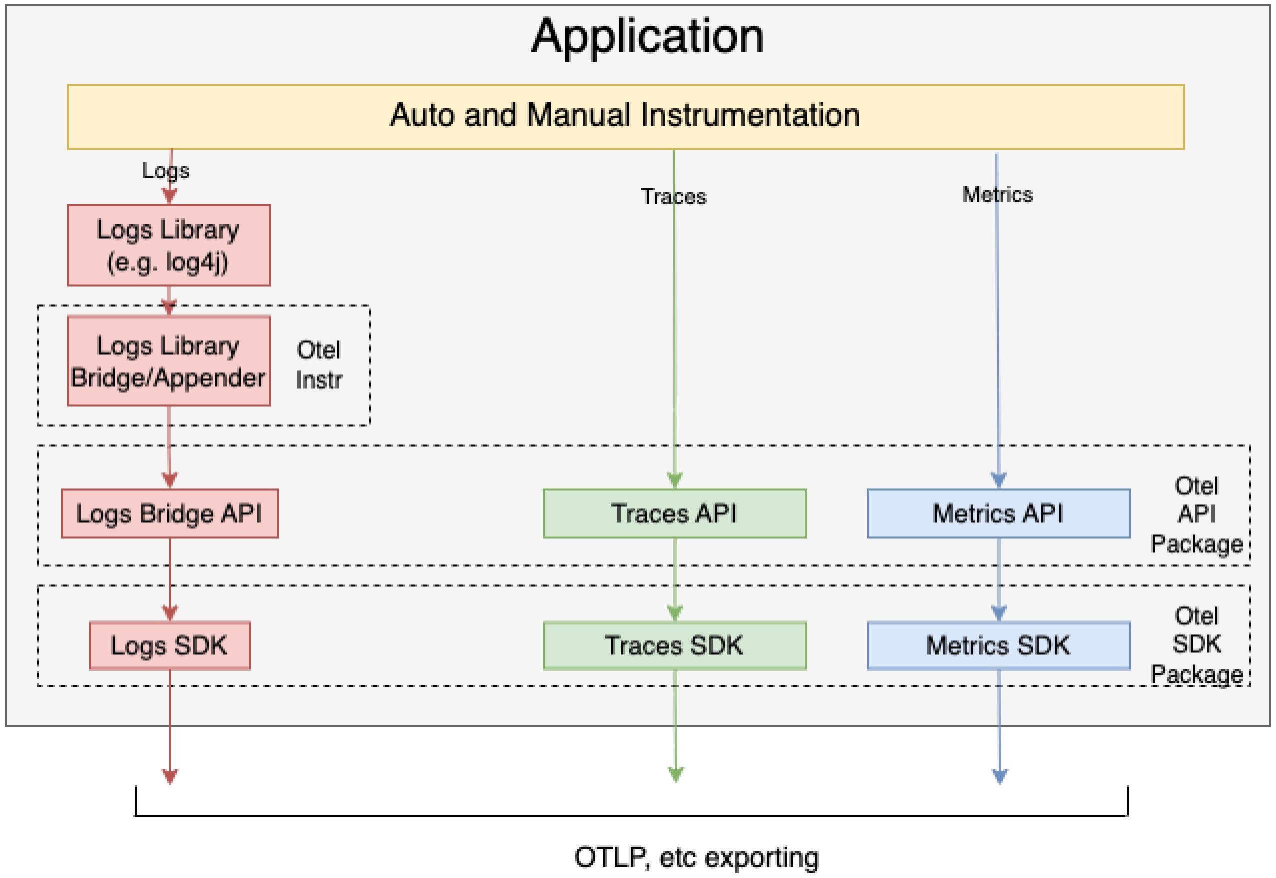




Software Observability – Industry Standards: OpenTelemetry (*Now*)







SoftScanner – OpenTelemetry Tracing with Zipkin

Thank you for your attention!
Any questions?