



SoftScanner: A Log-centric Software Analysis and Engineering Ecosystem

Bachar RIMA

**Equipe MAREL
En collaboration avec
Berger-Levrault**

06/10/2022



1. Introduction

- Background
- Context
- Problem Statement

2. SoftScanner

- Motivation
- Roadmap
- Feature Model
- Architecture
- Workflow
- Performance Monitoring Tracing Strategy
- Event-based GUI Widgets Tracing Strategy
- GUI Exploration

3. Current State of Affairs & Future Works

- *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 **traces** behind them in the console, allowing us to understand what happened.

```
1  public boolean addUser(int id, String name) {
2      // operation entry trace with provided data
3      // event = adding a user to a database
4      // context = id and name
5      System.out.println("Entered addUser() with id " + id + " and name " + name);
6
7      // creation of user
8      User user = new User(id, name);
9
10     // addition of user
11     ...
12
13     if (additionResult == true)
14         // trace in case of success
15         // event = successfully adding a user to a database
16         // context = user
17         System.out.println("Successfully added user " + user);
18     else
19         // trace in case of failure
20         // event = failed to add a user to a database
21         // context = user
22         System.out.println("Failed to add user " + user);
23 }
```

```
1  public void someOperation() {  
2      t1 = DateTime.Now();  
3  
4      // the actual operation implementation  
5      ...  
6  
7      t2 = DateTime.Now();  
8  
9      // execution time trace  
10     // event = execution success with execution time  
11     // context = t2 - t1 (the actual time)  
12     System.out.println("Finished executing someOperation after "  
13         + (t2 - t1) + " seconds");  
14 }
```

Problems with traditional printing statements

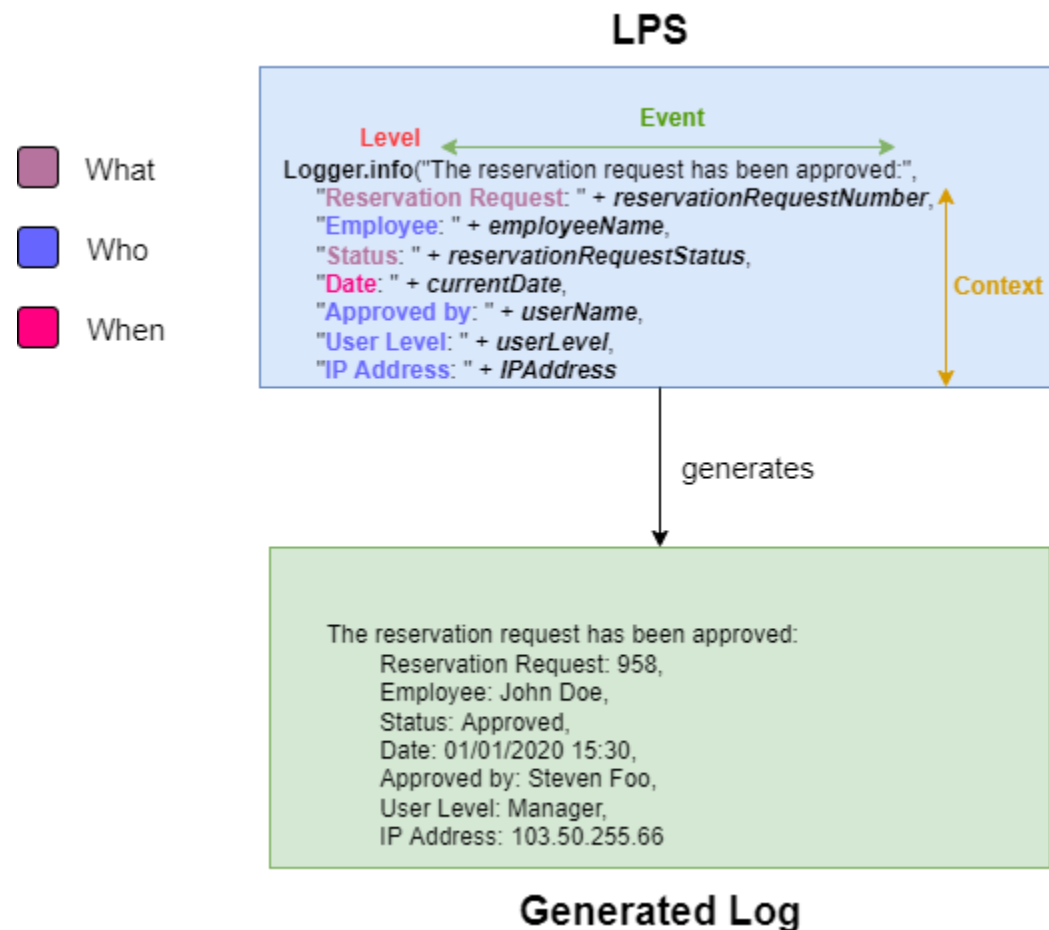
- **Ephemeral:** once the application is done executing, the printed statements are cleared from the console.
- **No verbosity/criticality control:** not all messages need to be printed for all events in all scenarios. For example, in a production environment we only want to see information about the 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 (LPSes)** into a project's code.

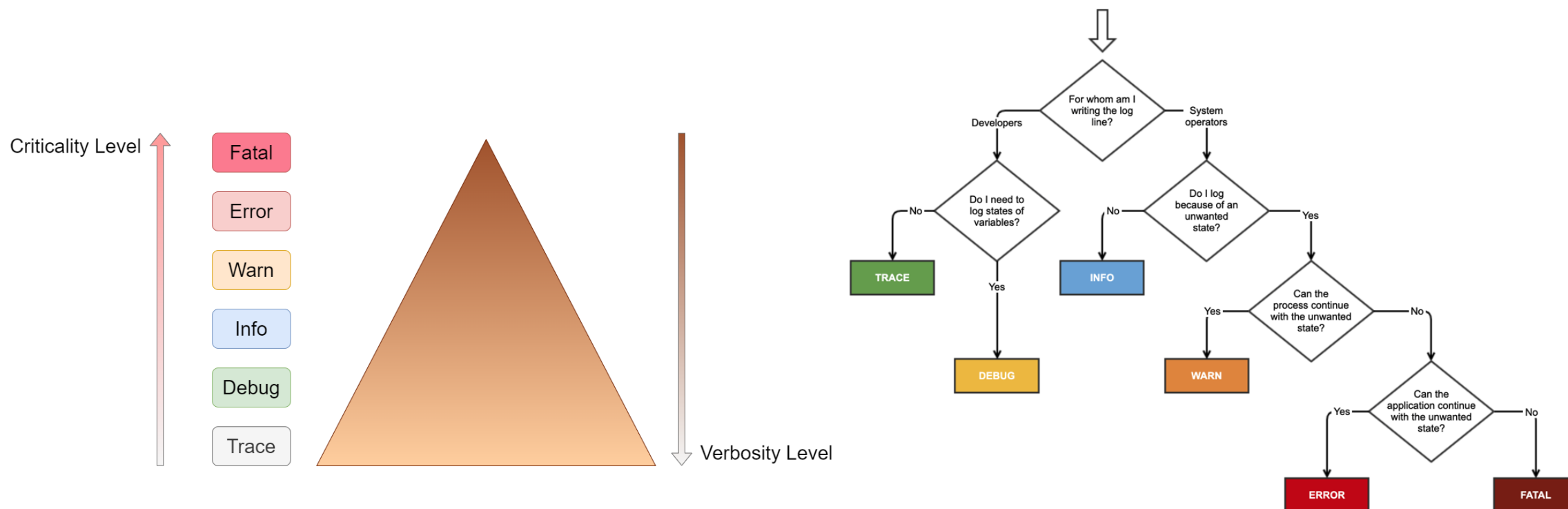
Context – What is logging?

- **Log Printing Statements (LPSes)**: tracing operations injected into a system's source code to extract information related to the fulfillment of a **tracing objective** 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 triggered (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.

Context – LPS vs. Log

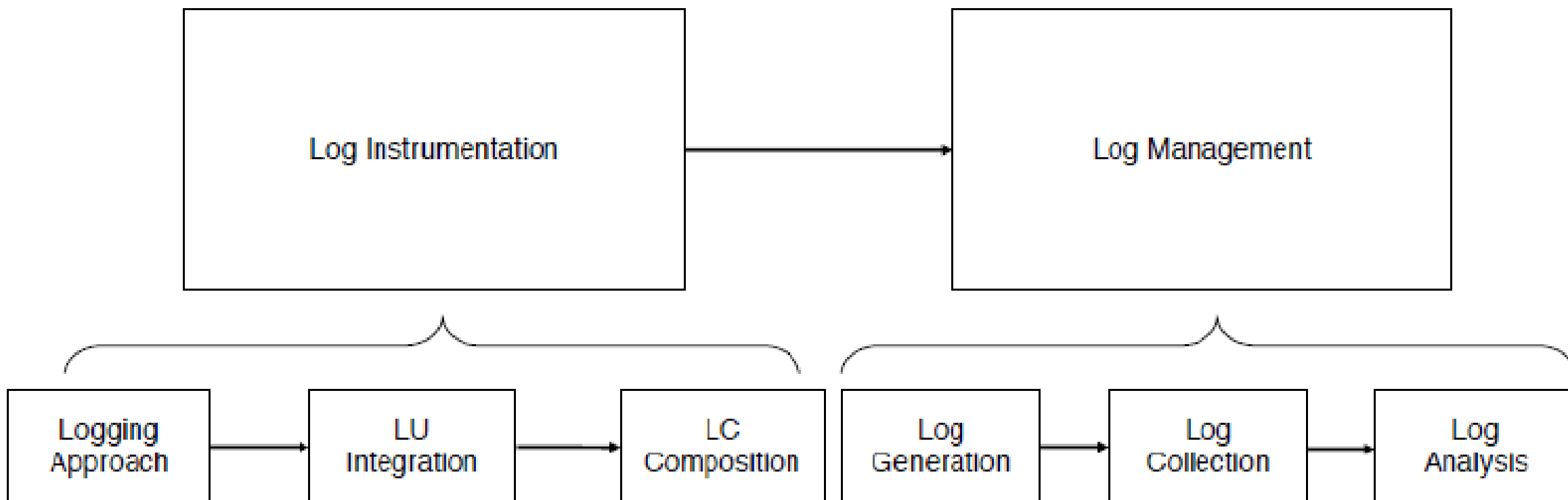


Context – LPS Criticality Level



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


Context – The Logging Process



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

Context – Logging Approaches (1)

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

 Application Developers

```

1 import org.apache.logging.log4j.LogManager;
2 import org.apache.logging.log4j.Logger;
3 class MyServer {
4     Logger logger = LogManager.getLogger(MyServer.class);
5     void authentication(Request req, ...) {
6         logger.info("Receive from client " + req.userName);
7         // actual authentication process ...
8     }
9     void reply(Response res, ...) {
10        logger.info("Send response to " + req.IP);
11    }
12 }
13 private void start() {
14     Server server = new Server();
15 }
        
```


MyServer.java

```

Apr 02, 2020 12:22:41 PM mycompany.MyServer Receive from client alice
...
Apr 02, 2020 12:22:42 PM mycompany.MyServer Send response to 192.168.0.1
...
Apr 02, 2020 12:22:50 PM mycompany.MyServer Receive from client bob
...
Apr 02, 2020 12:22:52 PM mycompany.MyServer Send response to 192.168.0.2
...
Apr 02, 2020 12:23:01 PM mycompany.MyServer Receive from client tom
...
Apr 02, 2020 12:23:02 PM mycompany.MyServer Send response to 192.168.0.2
...
        
```

(a) Using conventional logging for log instrumentation
(d) Outputted logs

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

 Application Developers

```

1 import org.apache.logging.log4j.LogManager;
2 import org.apache.logging.log4j.Logger;
3
4 @Around("execution(* MyServer.authentication(..))")
5 public Object logAround(ProceedingJoinPoint pjp, Request req) {
6     logger.info("Receive from client " + req.userName);
7     pjp.proceed();
8     logger.info("Send response to " + req.IP);
9 }
10
        
```

LogAspect.java

```

3 class MyServer {
4     void authentication(Request req, ...) {
5         // actual authentication process ...
6     }
7 }
        
```

MyServer.java

```

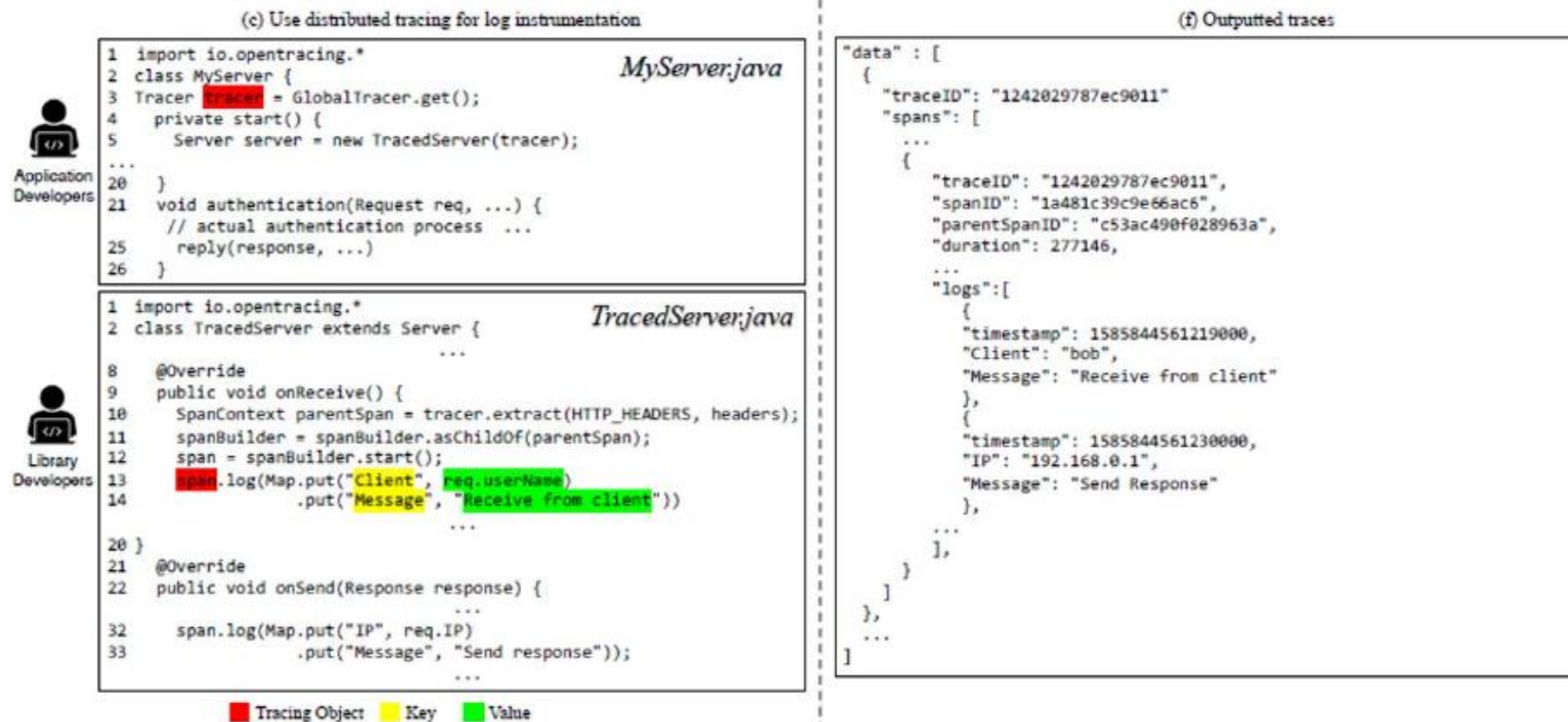
Apr 02, 2020 12:22:41 PM mycompany.MyServer Receive from client alice
...
Apr 02, 2020 12:22:42 PM mycompany.MyServer Send response to 192.168.0.1
...
Apr 02, 2020 12:22:50 PM mycompany.MyServer Receive from client bob
...
Apr 02, 2020 12:22:52 PM mycompany.MyServer Send response to 192.168.0.2
...
Apr 02, 2020 12:23:01 PM mycompany.MyServer Receive from client tom
...
Apr 02, 2020 12:23:02 PM mycompany.MyServer Send response to 192.168.0.2
...
        
```

(b) Using rule-based logging for log instrumentation
(e) Outputted logs

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

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*.

Context – Logging Approaches (3)

Dimension/Logging Approach	Conventional Logging	Rule-based Logging	Distributed Tracing
Who	SUS developers	SUS developers	Tracing Library developers
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

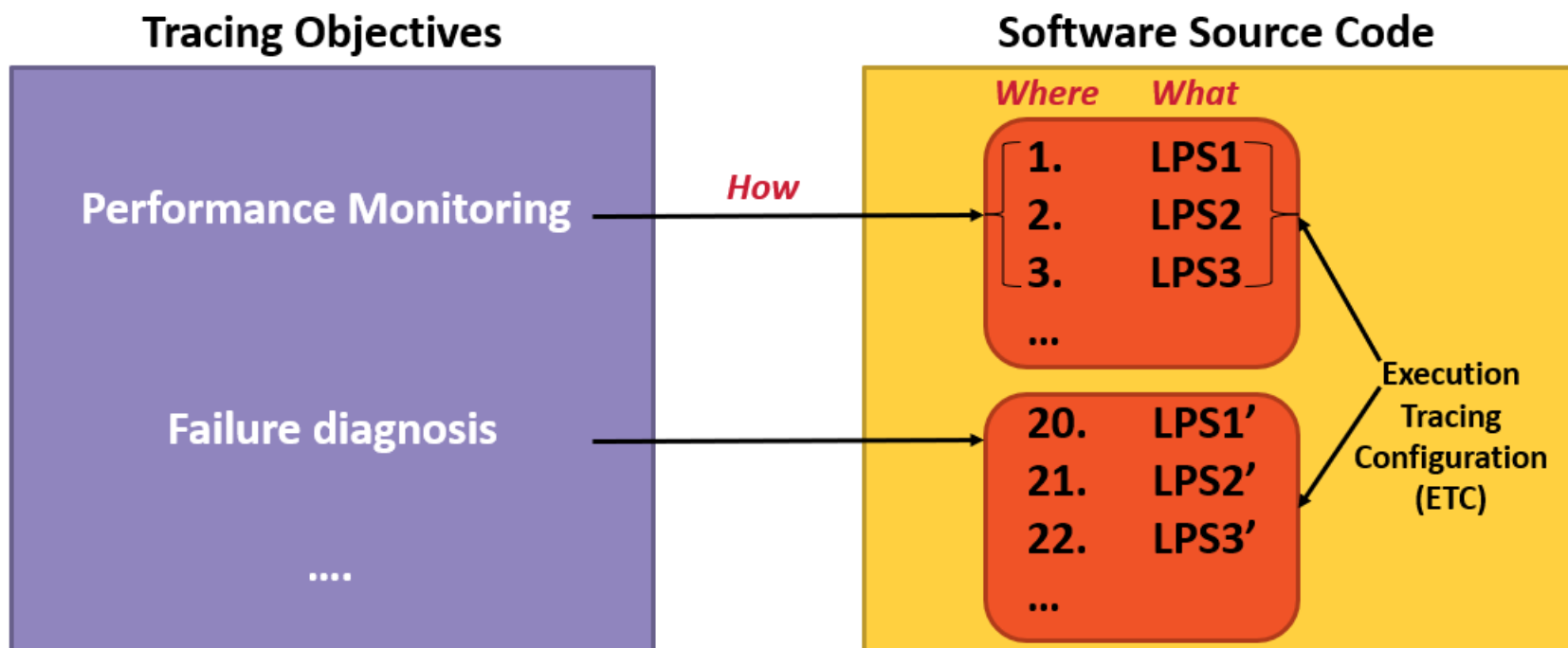
Chen, B. (2020). Improving the Logging Practices in DevOps.

- **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 tracing objectives?”*

- **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 in order to:
 1. Gain full observability 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?”*

Context – Logging Code Composition

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



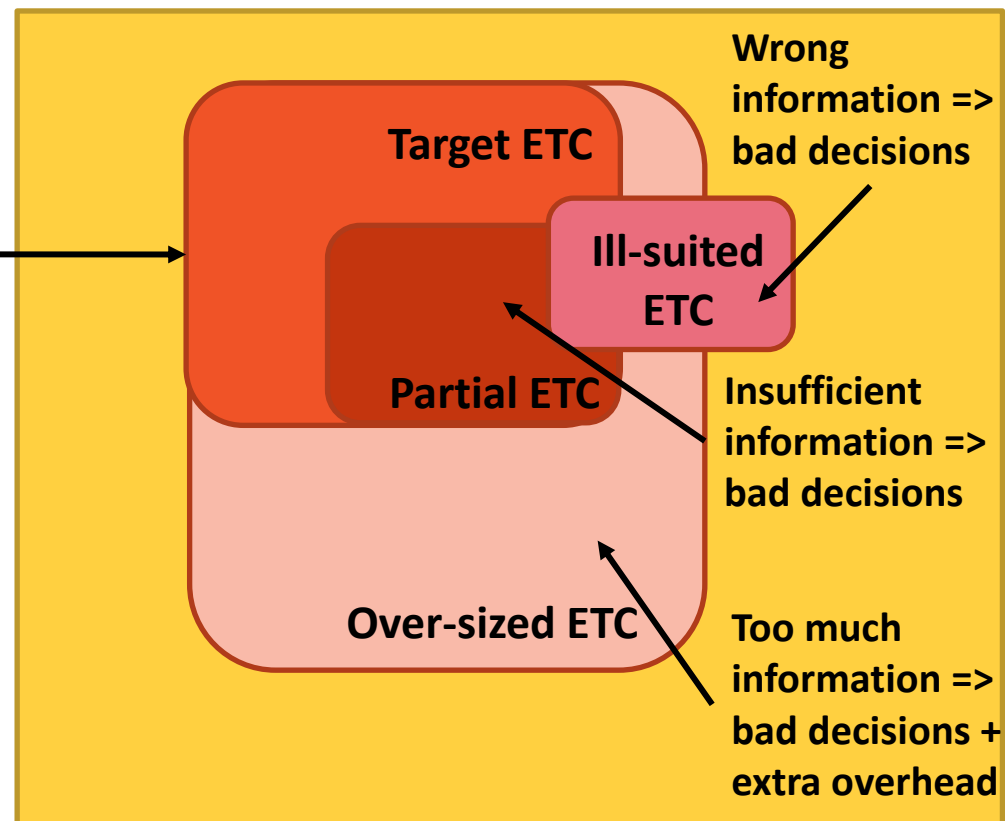
- **Very few systematic guidelines on log instrumentation exist**, even fewer are applied in practice.
- **Automated approaches for log instrumentation exist**, but they are mostly **partial** or **costly**:
 1. Handling one tracing objective at a time.
 2. Don't necessarily address all aspects of logging code composition:
 - ☐ Where to log only;
 - ☐ What to log only;
 - ☐ What log levels to use only;
 - ☐ Any incomplete combination of the above.

Problem Statement – Coverage Mismatch between Tracing Objectives and ETCs

Tracing Objectives

Performance Monitoring

Software Source Code



Problem Statement – Flexibility, Overlapping, and Redundancy of LPSes

Tracing Objectives

Performance Monitoring

Failure diagnosis

Software Source Code

ETC

Same LPSes =>
redundancy and
overlapping which
can lead to
overhead

Same LPS location,
yet different LPS
content => lack of
flexibility in an
LPS' coverage

ETC

- **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 tracing objective at a time** → need for logging support for multiple tracing objectives 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 tracing objective by an ETC** may result in bad decisions, and possibly extra overhead → need for optimization mechanisms to generate ETCs that optimally cover their tracing objectives
- **Redundant and overlapping LPSes** may lead to overhead and lack of flexibility → need for optimization mechanisms to handle redundant and overlapping LPSes

- 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-tracing-goal-based log instrumentation approach → need for logging support for multiple tracing objectives at once
- SoftScanner provides different logging strategies to build ETCs for specific tracing objectives, with different criticality level output filters, while establishing a bi-directional traceability link between the LPSes 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 ETCs that optimally cover their tracing objectives
 - ❑ need for optimization mechanisms to handle redundant and overlapping LPSes

Axis 1: Tracing Objectives & ETCs

- Make a feature model of all possible tracing objectives
- Select tracing objectives of interest
- Implement at least one logging strategy for each tracing objective of interest

Axis 2: Static optimization of ETCs

- **Goal:** Perform static generation of optimal ETCs for multiple tracing objectives at once
- **Approach:** Implement different optimization techniques that balance the trade-off between reducing ETC costs and maximizing its tracing goal's degree of fulfillment

Axis 3: Dynamic optimization of ETCs

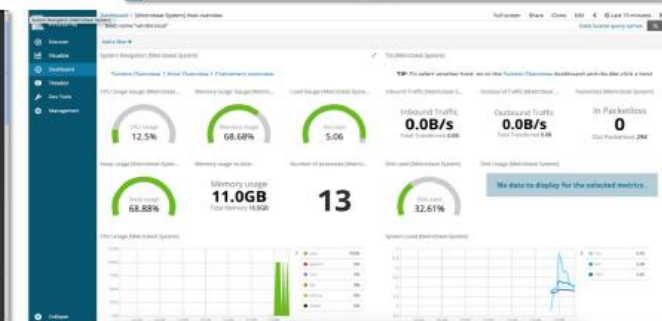
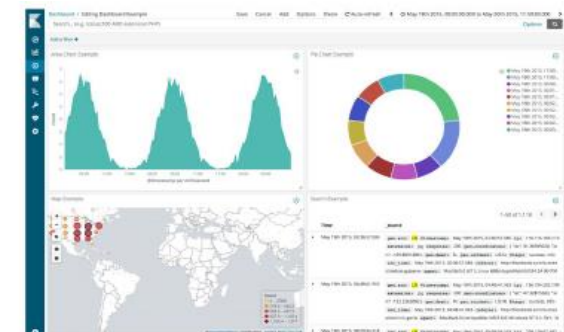
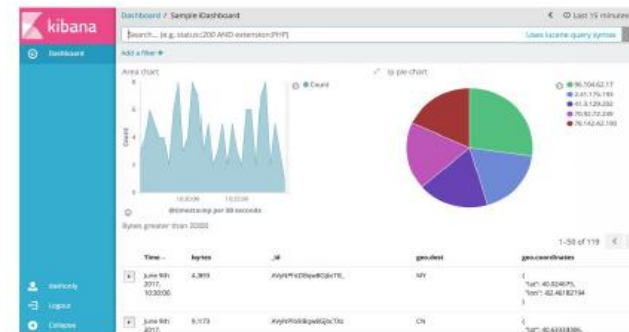
- **Goal:** Allow a dynamic generation of ETCs where data collection points can be enabled, disabled, and/or defined on the run to fulfill a given set of tracing goals
- **Approach:** Apply information dynamic analysis and coding techniques that allow the dynamic enabling, disabling, and definition of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

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

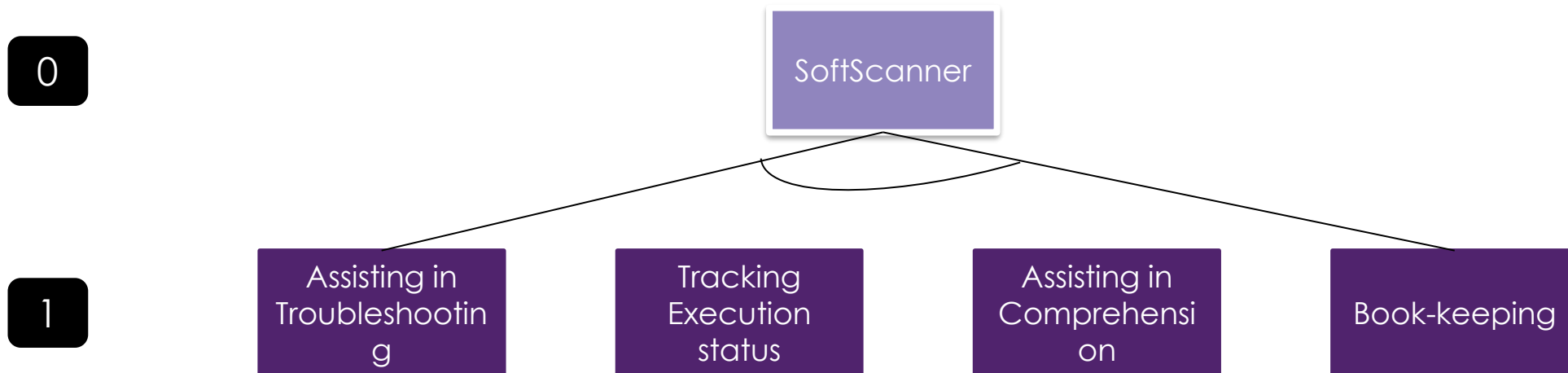
- **Goal:** Make the approach platform-independent by generalizing concepts tackled in the previous axes and making them model-driven using MDE
- **Approach:** Define and reuse a set of metamodels covering the relevant domains (source code, ETCs, LPSes, tracing objectives, ...) as well as the necessary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

Axis 5: Interactive visual platform for ETC adaptation

- **Goal:** Allow operators to directly adapt ETCs with respect to the desired tracing 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

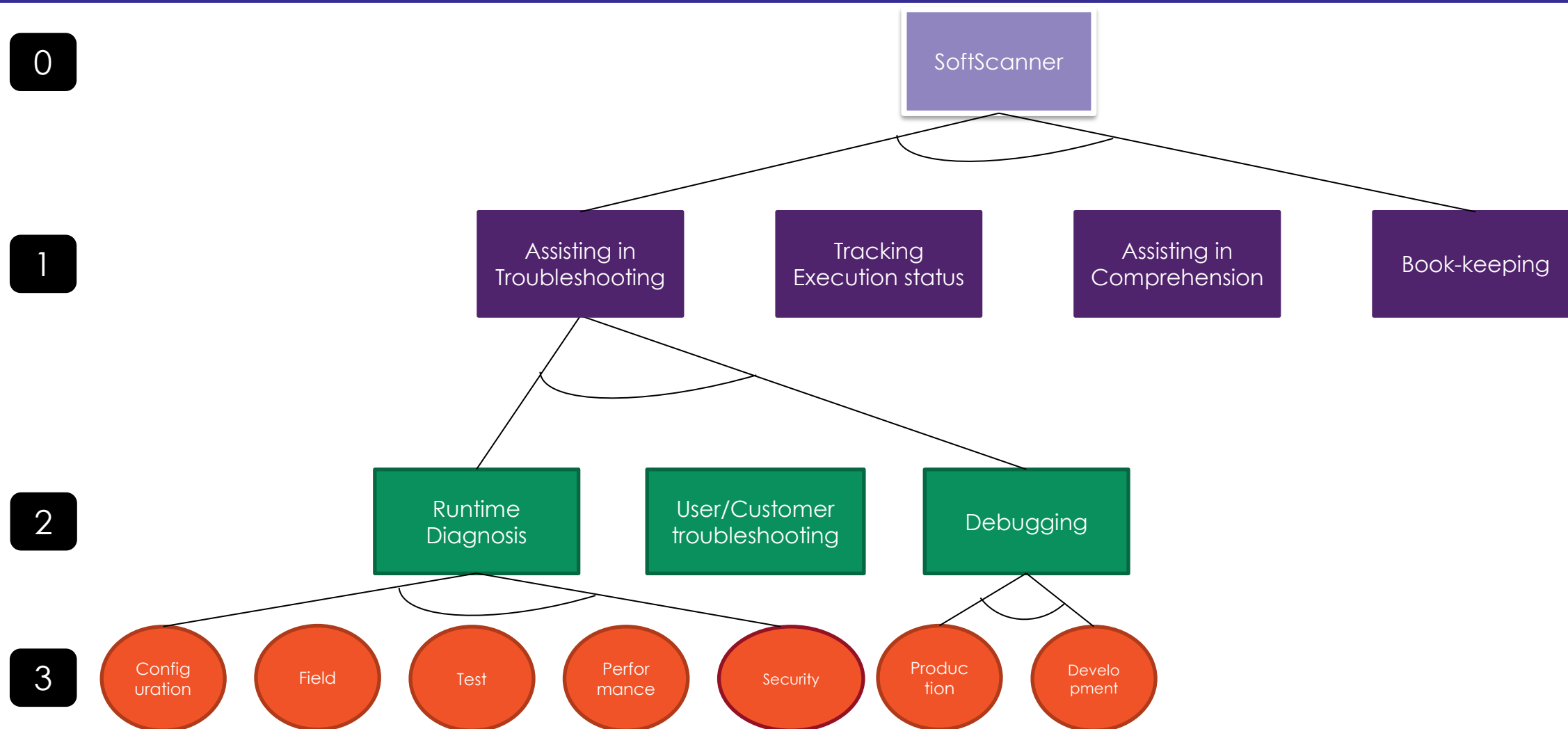


SoftScanner – Feature Model (1)



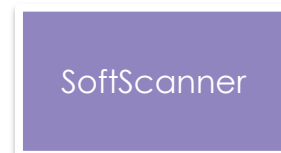
Based on *H. Li et al. (2020). A Qualitative Study of the Benefits and Costs of Logging from Developers' Perspectives.*

SoftScanner – Feature Model (2)

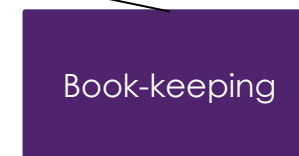
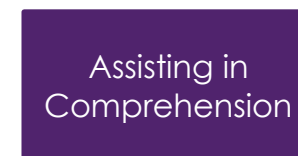
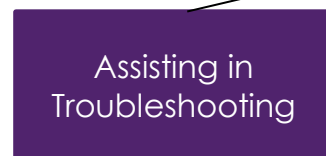


SoftScanner – Feature Model (3)

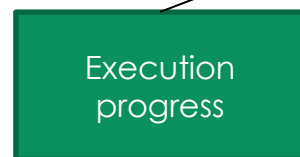
0



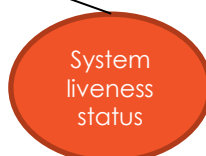
1



2



3



SoftScanner – Feature Model (4)

0

SoftScanner

1

Assisting in
Troubleshooting

Tracking
Execution
status

Assisting in
Comprehension

Book-
keeping

2

System
comprehension

Assisting in
software
development

3

Code
familiarization

System
runtime
familiariza-
tion

Real vs.
Expected
system
behavior
verification

Detecting
code
smells

Suggesting
better coding
practices

4

Major
runtime
events

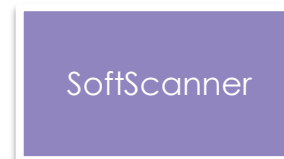
Use-case
Scenario
dynamics

Logging
events
contextu-
alization

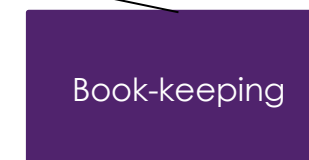
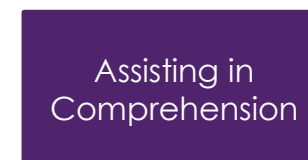
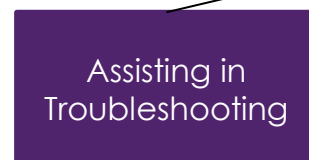
Internal state
communication

SoftScanner – Feature Model (5)

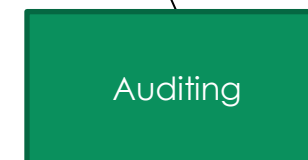
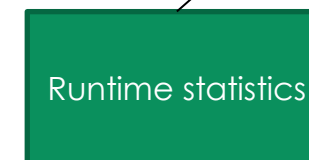
0



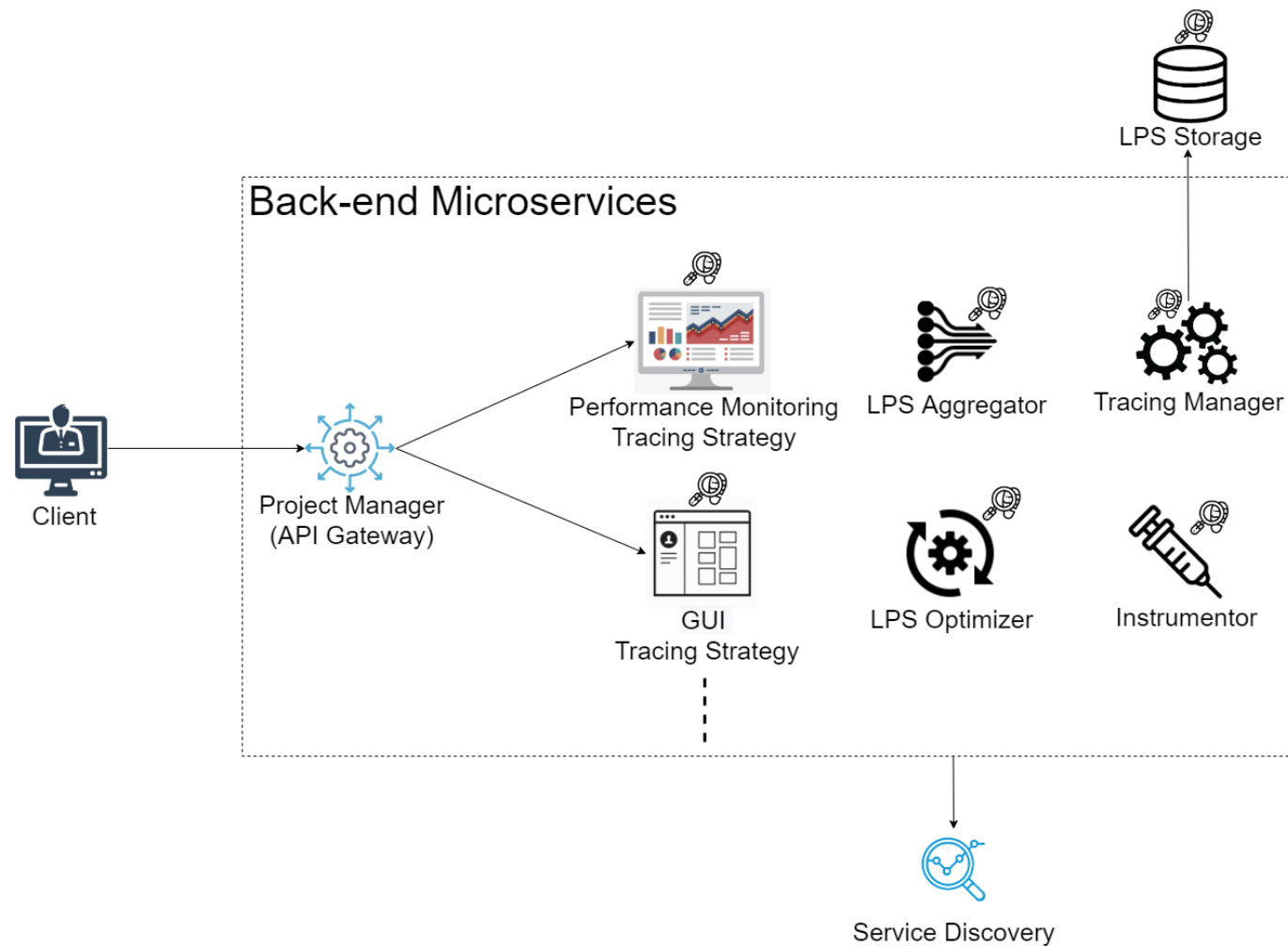
1



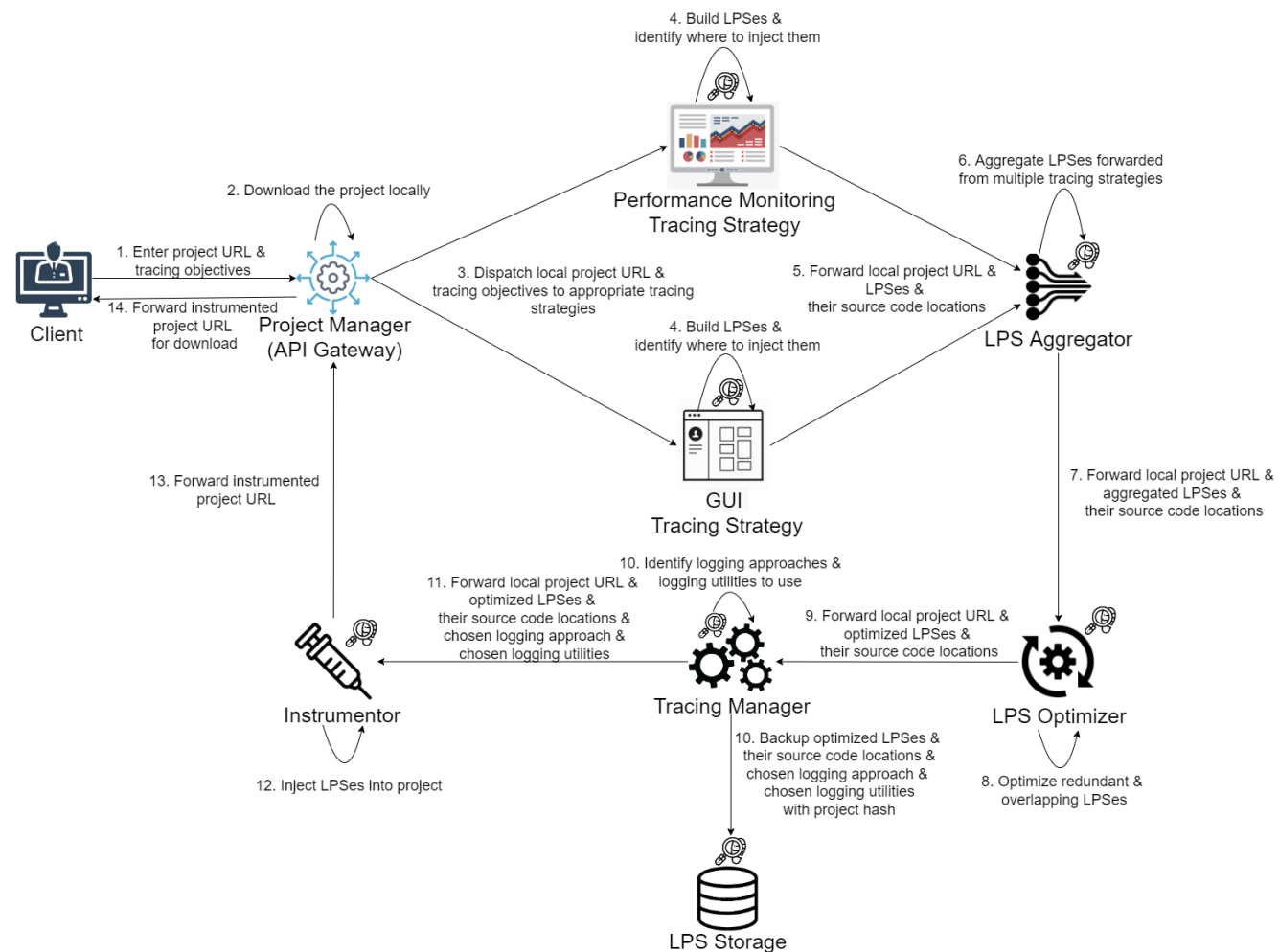
2



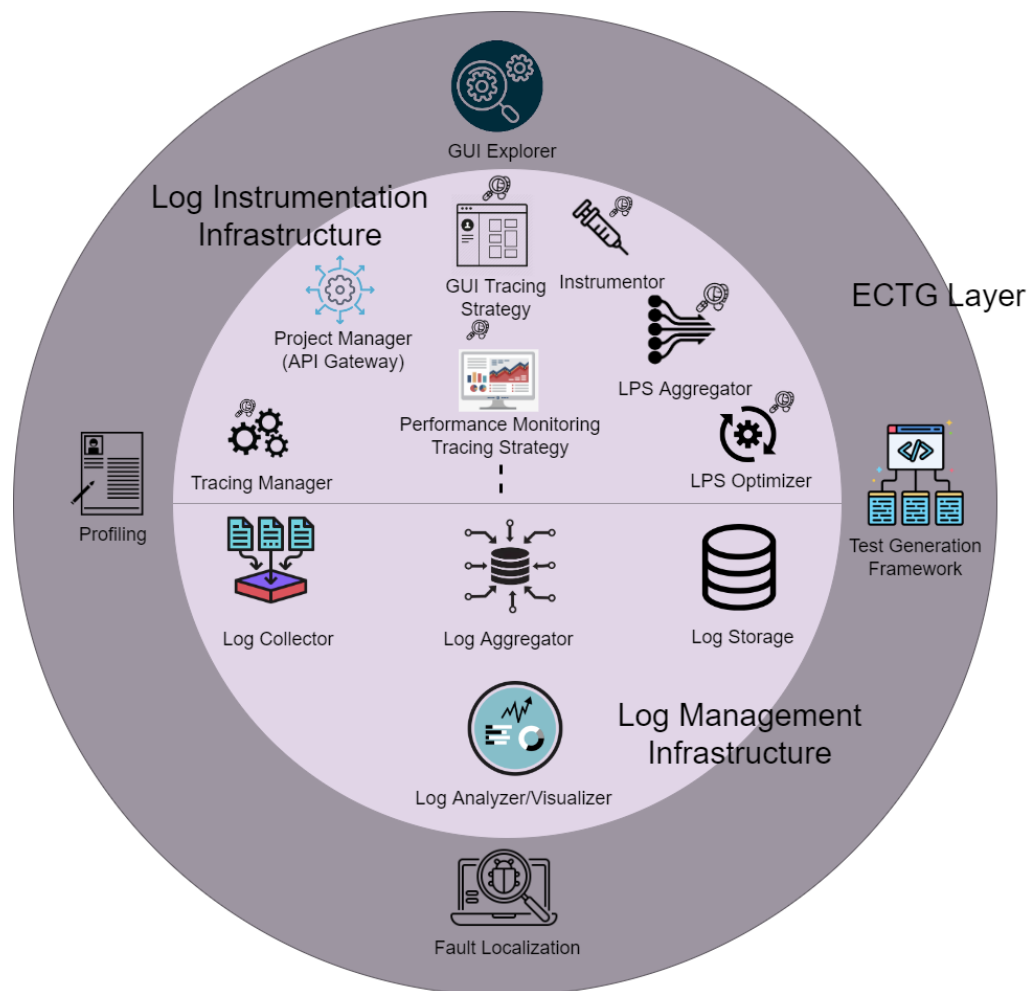
SoftScanner – Logging Instrumentation Architecture



SoftScanner – Logging Instrumentation Workflow

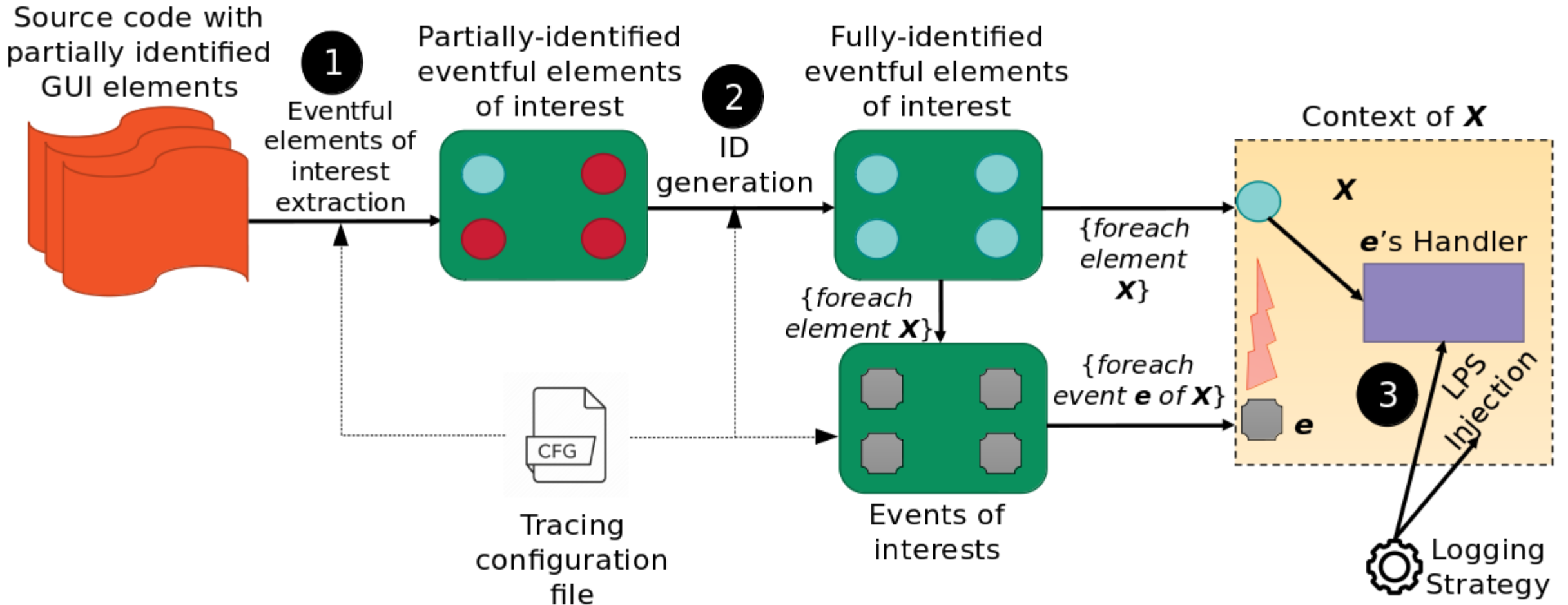


SoftScanner – Ecosystem



- **Goal:** provide a tracing mechanism for GUI widgets of Angular projects.
- **Methodology:** given a set of target events, automatically trace 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 Tracing (2)

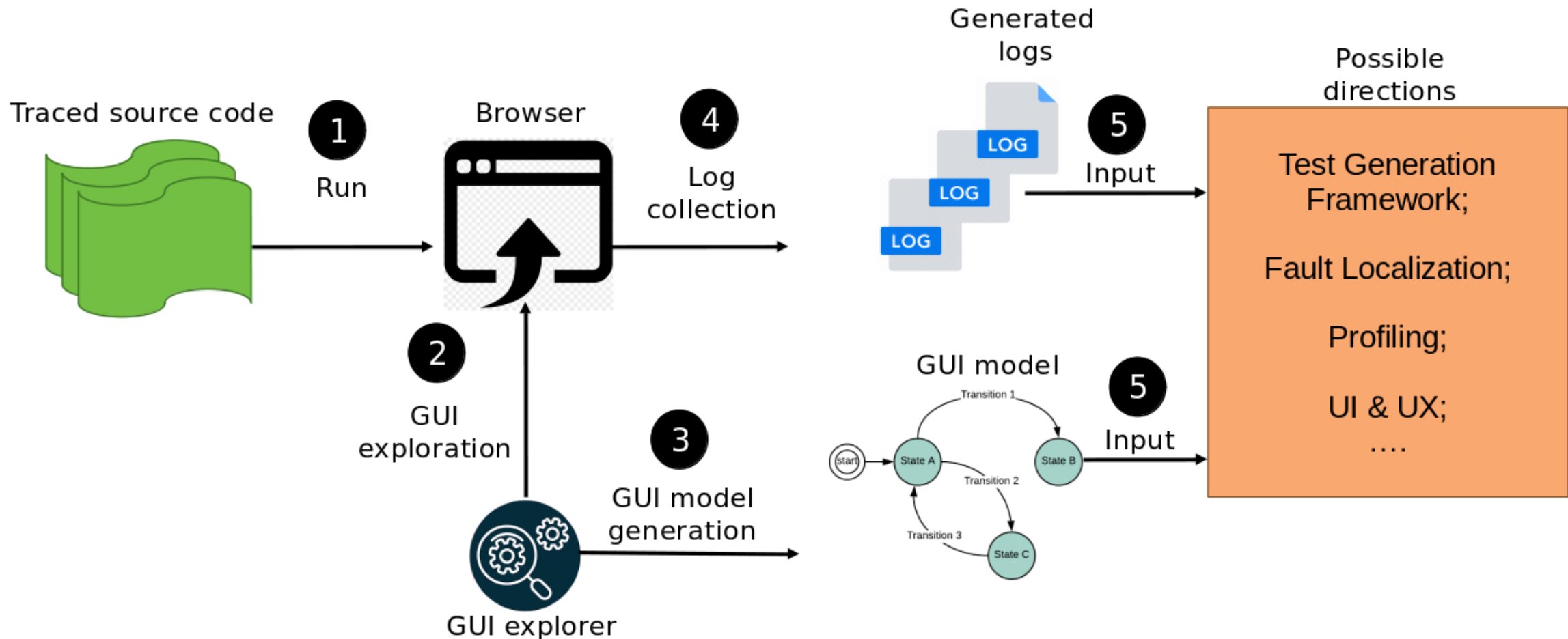


SoftScanner – Event-based GUI Widgets Tracing (3)



- **Based on:**
 1. **GUI Ripping** (*Memon A. et al. (2003). GUI ripping: Reverse Engineering of Graphical User Interfaces for Testing*)
 2. **RIPuppet** (*Osorio, J. M. D. (2019). Automated GUI ripping for web applications*)
- **Goal:** model a web application's GUI and leverage it for different software analysis and engineering tasks, including tracing, testing, profiling, UI & UX, ...
- **Methodology:**
 1. Construct a **GUI model** (*state machine*) consisting of **GUI states** and **transitions** using a **DFS graph exploration** algorithm and Selenium to simulate user interactions
 2. **GUI state:** a set of **executable** widgets (i.e., GUI widgets tagged with an executable heuristic and that the user can interact with)
 3. **GUI state transition:** a source state, a destination state, the event causing the transition, and the source state widget where the event has been triggered

SoftScanner – GUI Exploration (2)








SoftScanner – GUI Exploration (3)



Current State of Affairs (1)

Service/Theme	Status	Notes
Performance Monitoring Tracing	✓	Optimization Phase
GUI Tracing	✓	<ul style="list-style-type: none"> Optimization Phase Extensibility to React
Instrumentor	✓	Optimization Phase
Project Manager	✓	<ul style="list-style-type: none"> More test apps required Feature addition phase
GUI Explorer	✓	<ul style="list-style-type: none"> More test apps required Feature addition phase
LPS Aggregator	⌚	-
LPS Optimizer	⌚	-
Tracing Strategy Manager	⌚	-

Current State of Affairs (2)

Service/Theme	Status	Notes
Test Generation Framework		More research required
Profiling		More research required
UI & UX		More research required
Fault Localization		More research required
Tracing for Security		More research required

Future Works – Event-based GUI Widgets Tracing & GUI Exploration

Event-based GUI Widgets Tracing

- Increase and diversify our test applications dataset to extend the approach's generalizability and exhaustivity
- Extend the approach over other web SPAs (**React** (*in progress*), **Vue**, **Ember**, **Svelte**, ...)
- Generalize the approach using MDE to make it reusable, generic and configurable across different platforms and frameworks.

GUI Exploration

- Semi-automated form-filling for problematic scenarios (e.g., authentication-requiring views)
- Identify form validation mechanisms and update the exploration algorithm accordingly
- Handle ads and pop-ups
- Experiment with different persistence formats (JSON schemas, graph-oriented databases, ...)
- Experiment with the resulting GUI models and other artifacts for software analysis tasks (tests, profiling, UX & UI)

Thank you for your attention!
Any questions?

