

# SoftScanner: A Semi-Automatic Software Tracing & Testing Infrastructure



Bachar RIMA

Equipe MAREL


*En collaboration avec Berger-Levrault*

16/11/2021



# Motivation (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 a website prefer a specific product over another?
- Etc.

First answer :  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.

# Motivation (2)

```
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 }
```

# Motivation (3)

```
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 }
```

# Motivation (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 printed messages need to be displayed 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.

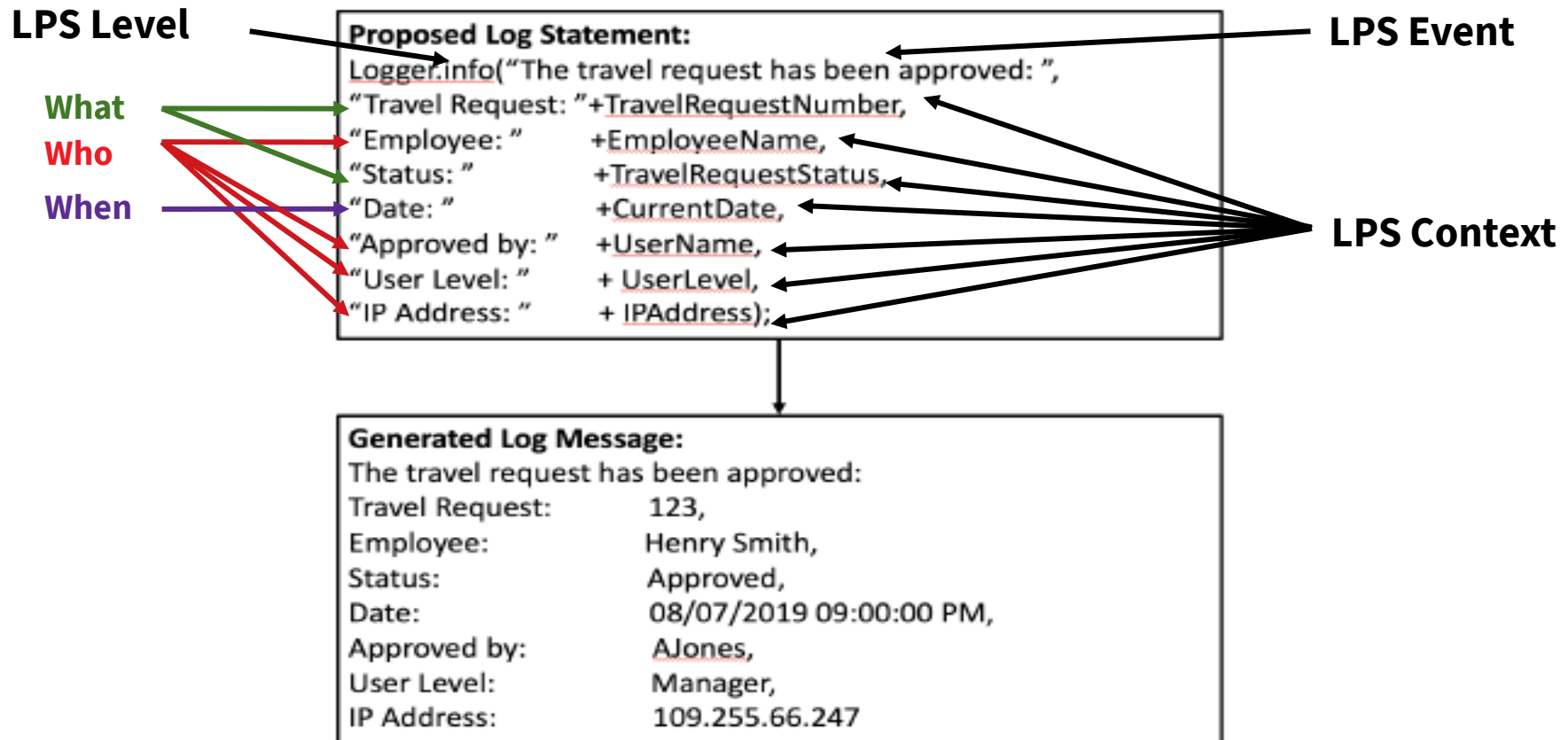
No printed message distribution : printed statements are only displayed in the standard output/error console. They cannot be dispatched to other destinations simultaneously.

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

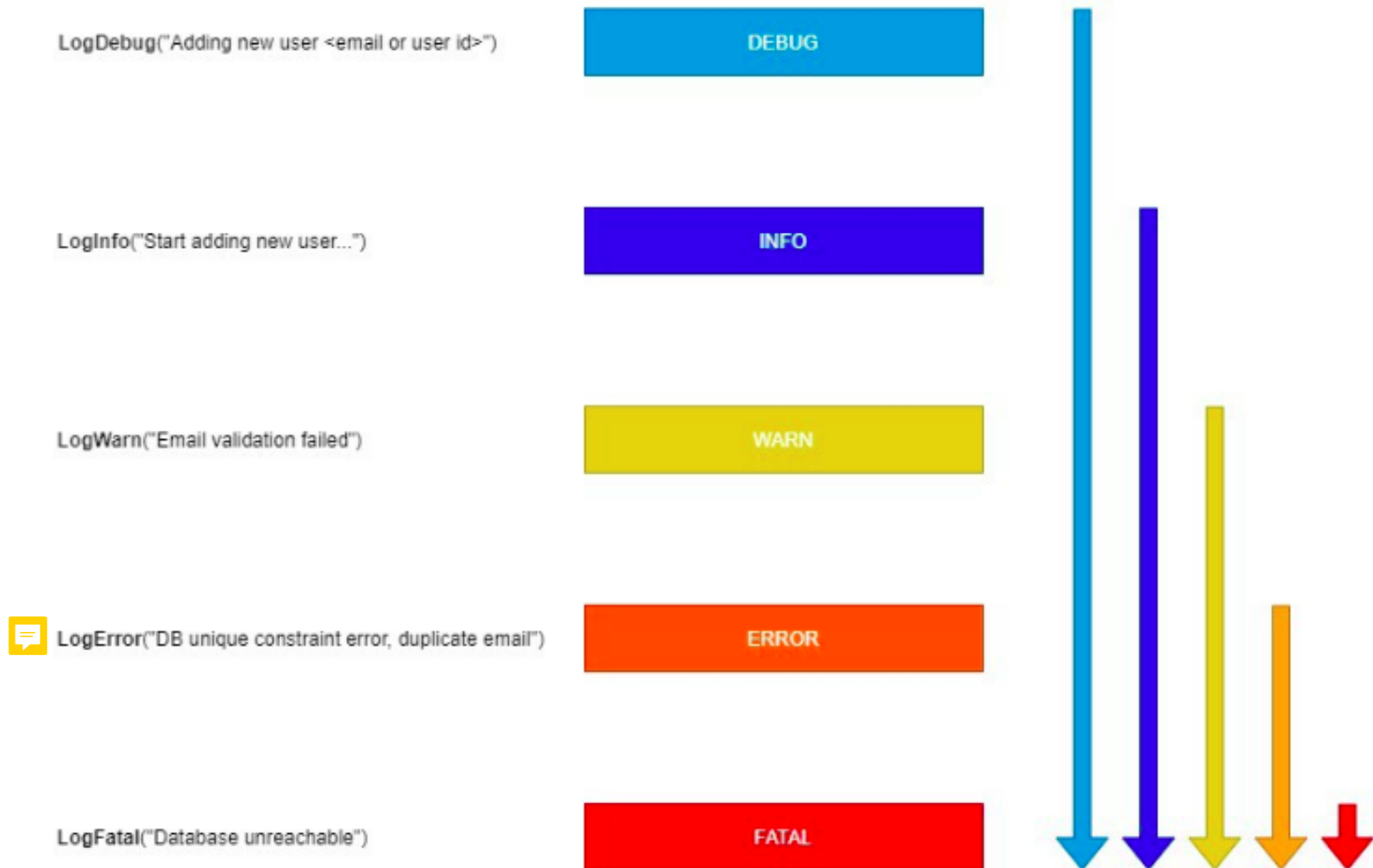
# What is Logging?

- 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 which consists of four main parts:
  - **Who**: Who triggered the event.
  - **When**: When was the event triggered.
  - **Where**: Where the event was triggered (e.g., GUI widget).
  - **What**: What event information to retrieve (event type and other event-related data).
- LPS level: the criticality of the generated log, controlling what kind of information will be included in each part of the LPS event.

# LPS vs. Log

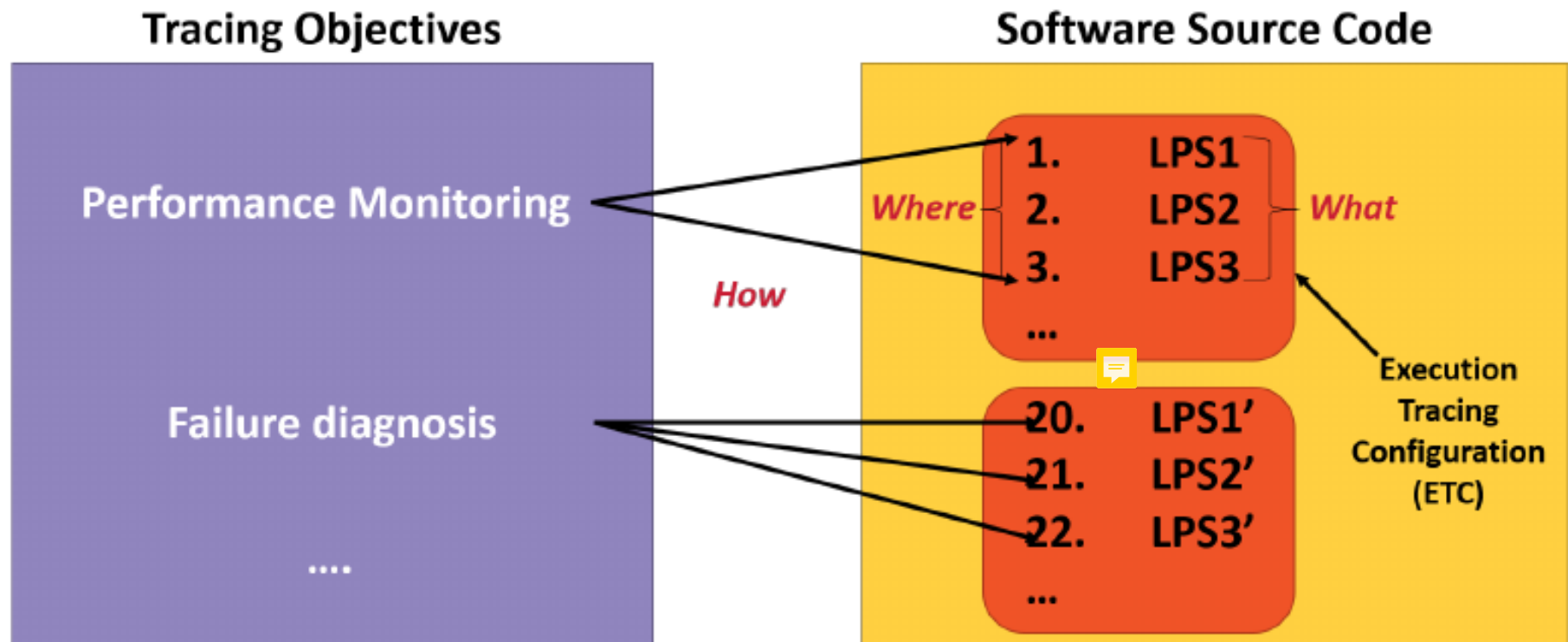


# LPS Criticality Level

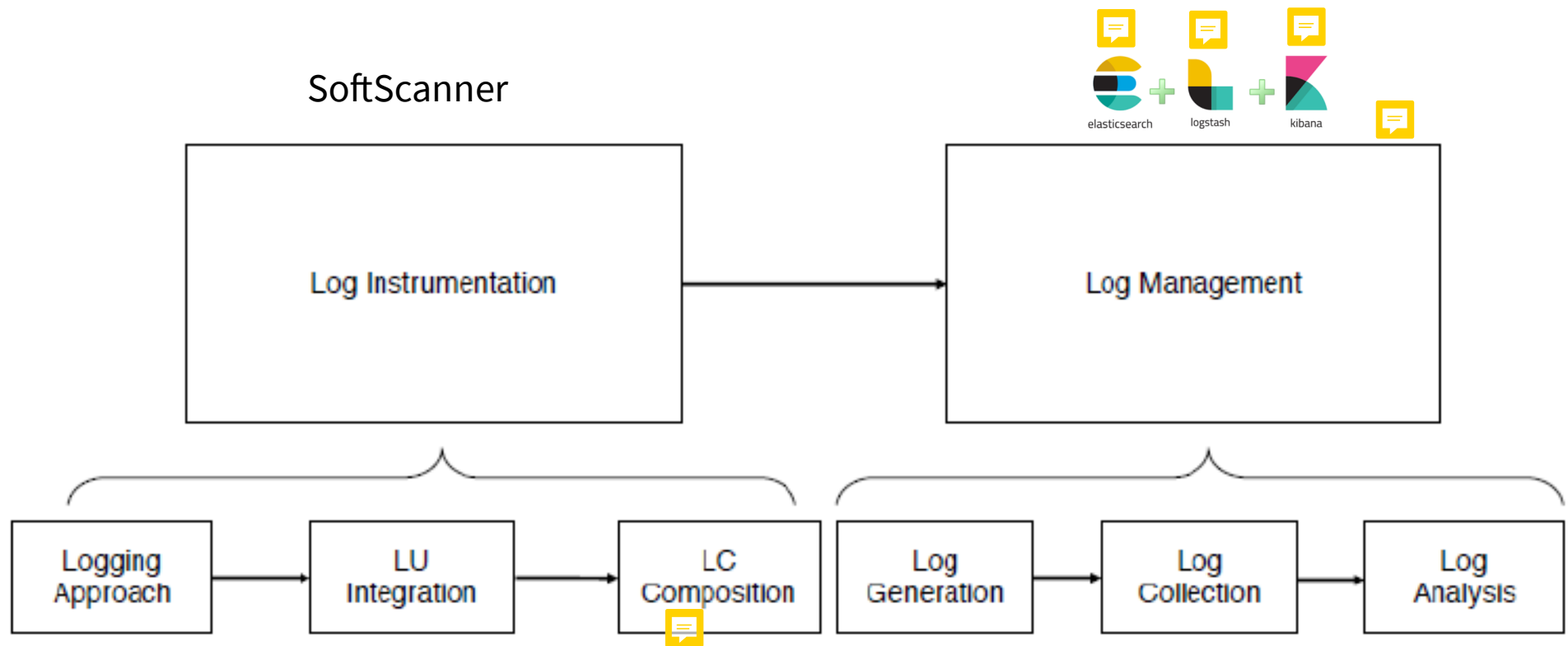




# Tracing Objectives & ETCs



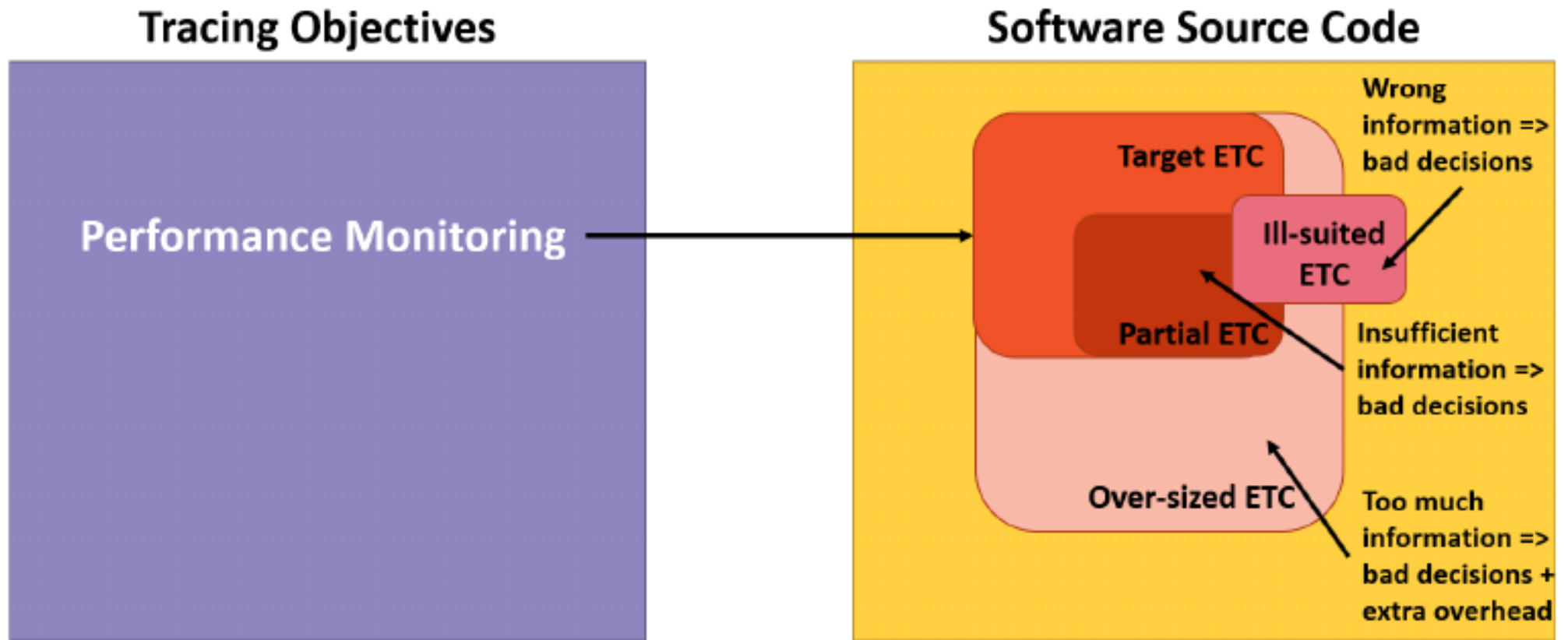
# The Logging Process



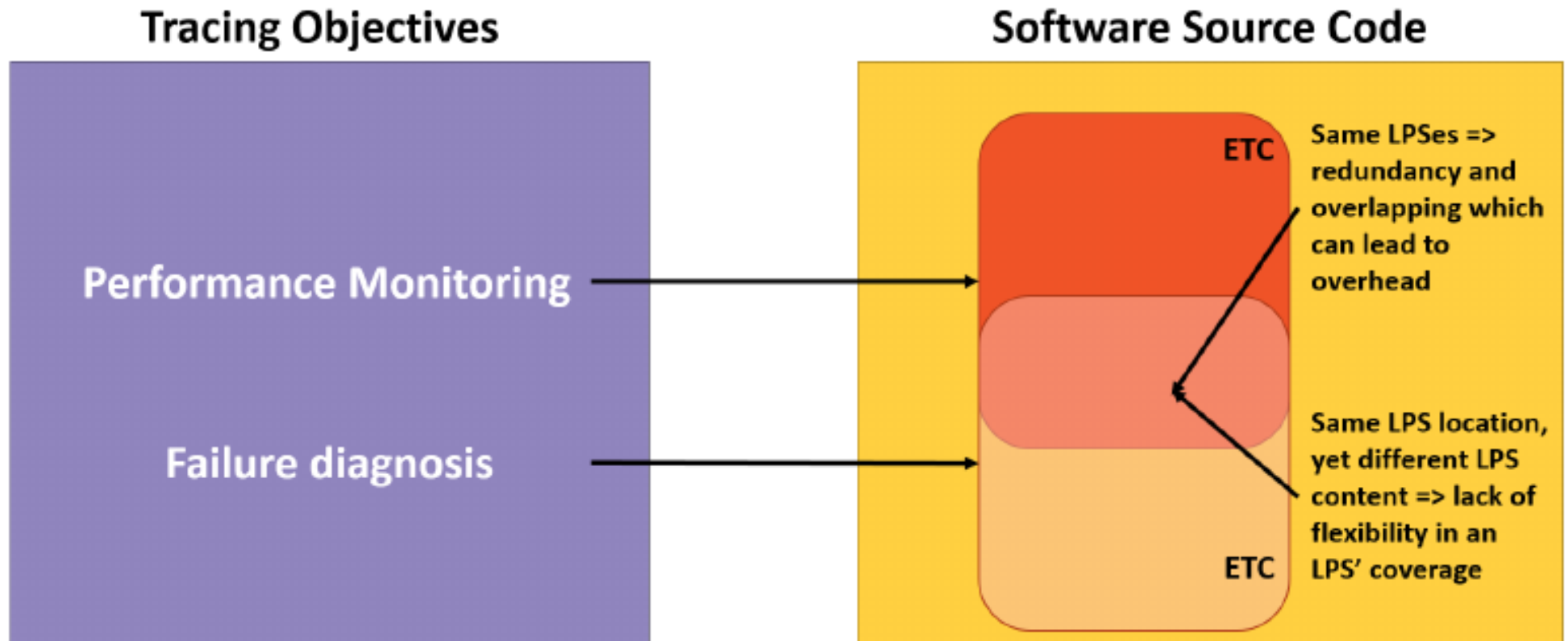
# Log Instrumentation Problems (1)

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

# Log Instrumentation Problems (2)



# Log Instrumentation Problems (3)



# Why SoftScanner? (1)

- Very few systematic guidelines on log instrumentation, and even fewer are applied in practice → need for more automation to reduce developer effort and error.
- Automated solutions for log instrumentation exist, but they are mostly partial or costly:
  - Handling one tracing objective at a time → need for logging support for multiple tracing objectives at once.
  - need for an approach covering all aspects of LPS composition (where, what, how, and with what level).
- Non-optimal coverage for a tracing objective by an ETC may result in bad decisions, and possibly extra overhead → need for optimisation mechanisms to generate ETCs that optimally cover their tracing objectives.
- Redundant and overlapping LPSes may lead to overhead and lack of flexibility → need for optimisation mechanisms to handle redundant and overlapping LPSes.

# Why SoftScanner? (2)

- 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 tracing goal strategies to create LPSes for specific tracing goals and with different criticality level outputs, but also to identify their source code locations → need for an approach covering all aspects of LPS composition (where, what, how, and with what level).
- SoftScanner provides an LPS optimizer component to deal with redundancy, overlapping, covering mismatch, and other LPS-related optimization issues →
  - need for optimisation mechanisms to generate ETCs that optimally cover their tracing objectives
  - need for optimisation mechanisms to handle redundant and overlapping LPSes.

# SoftScanner Original Roadmap (1)

## Axis 1: Tracing Objectives & ETCs

- Make a feature model of all possible tracing objectives.
- Select tracing objectives of interest.
- Define ETCs (through logging strategies) for each tracing objective of interest.

## Axis 2: Static optimization of ETCs

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



# SoftScanner Original Roadmap (2)

## Axis 3: Dynamic optimization of ETCs

- Goal: Allow a dynamic generation of ETCs, where data collection points can be defined on the run to fulfil a given set of tracing goals.
- Approach: Apply information coding techniques that allow the dynamic enabling/disabling of LPSes' execution.

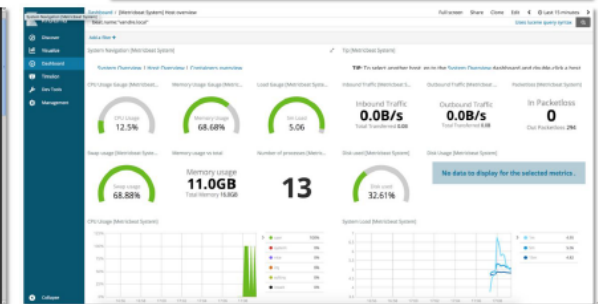
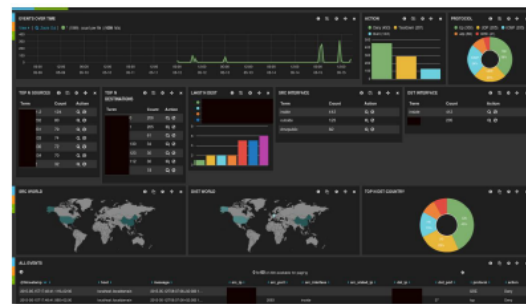
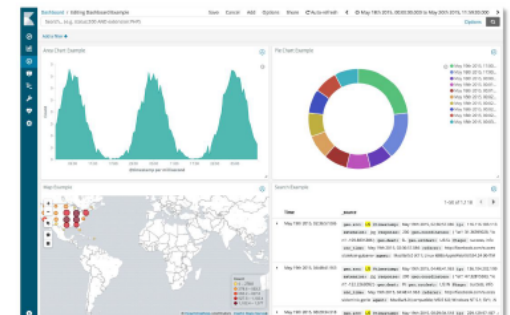
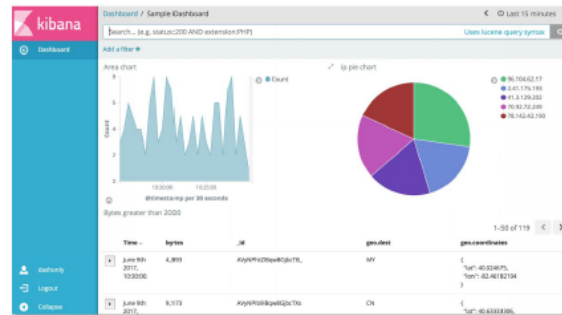
## Axis 4: Model-Driven-Engineering-based generation of ETCs

- Goal: Make the approach platform-independent, by generalizing the concepts tackled in the previous axes and making them model-driven, using MDE.
- Approach: Define/reuse a set of metamodels covering the domains addressed in this project (Source code, ETCs, LPSes & cost computation, tracing goals & their degrees of fulfillment, etc.).

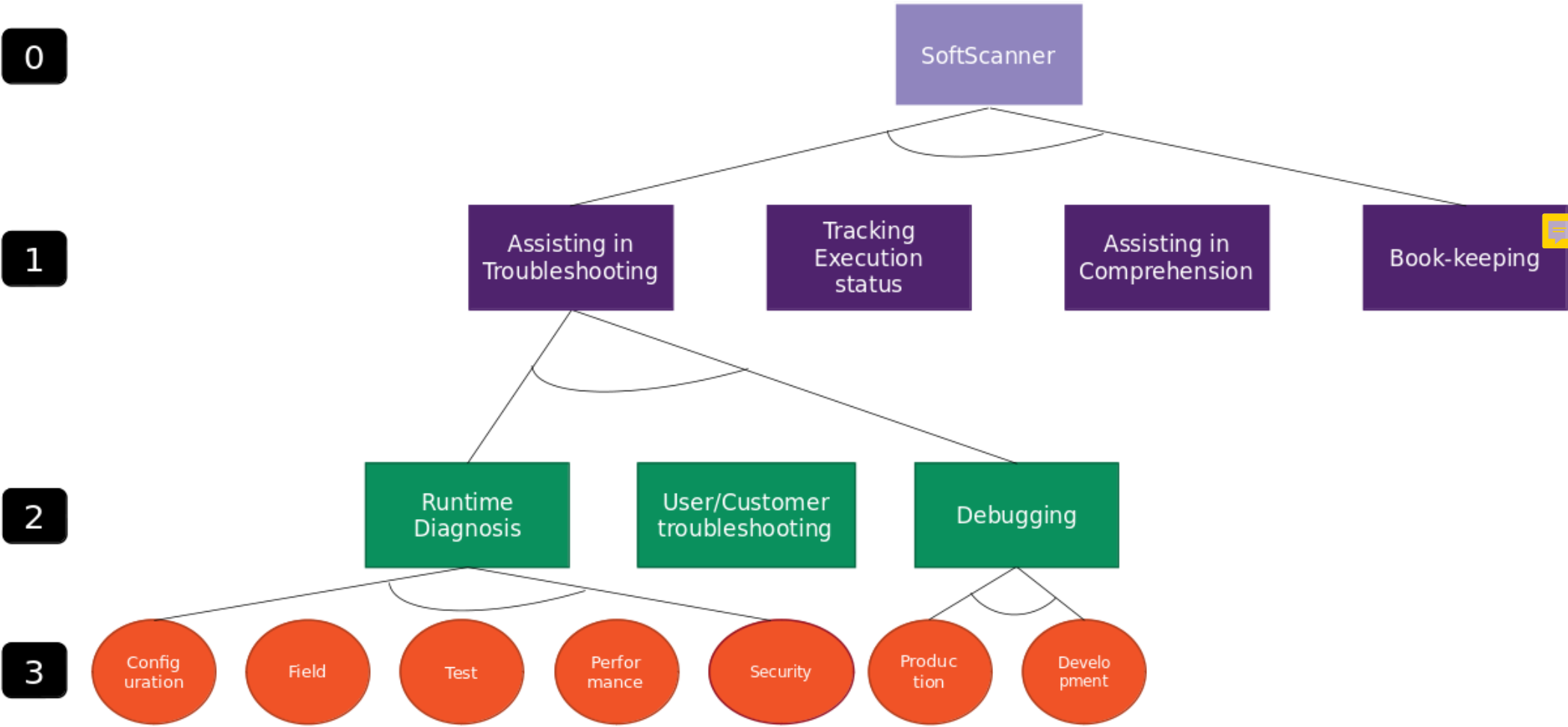
# SoftScanner Original Roadmap (3)

## Axis 5: Visual Interaction Platform for ETC adaptation

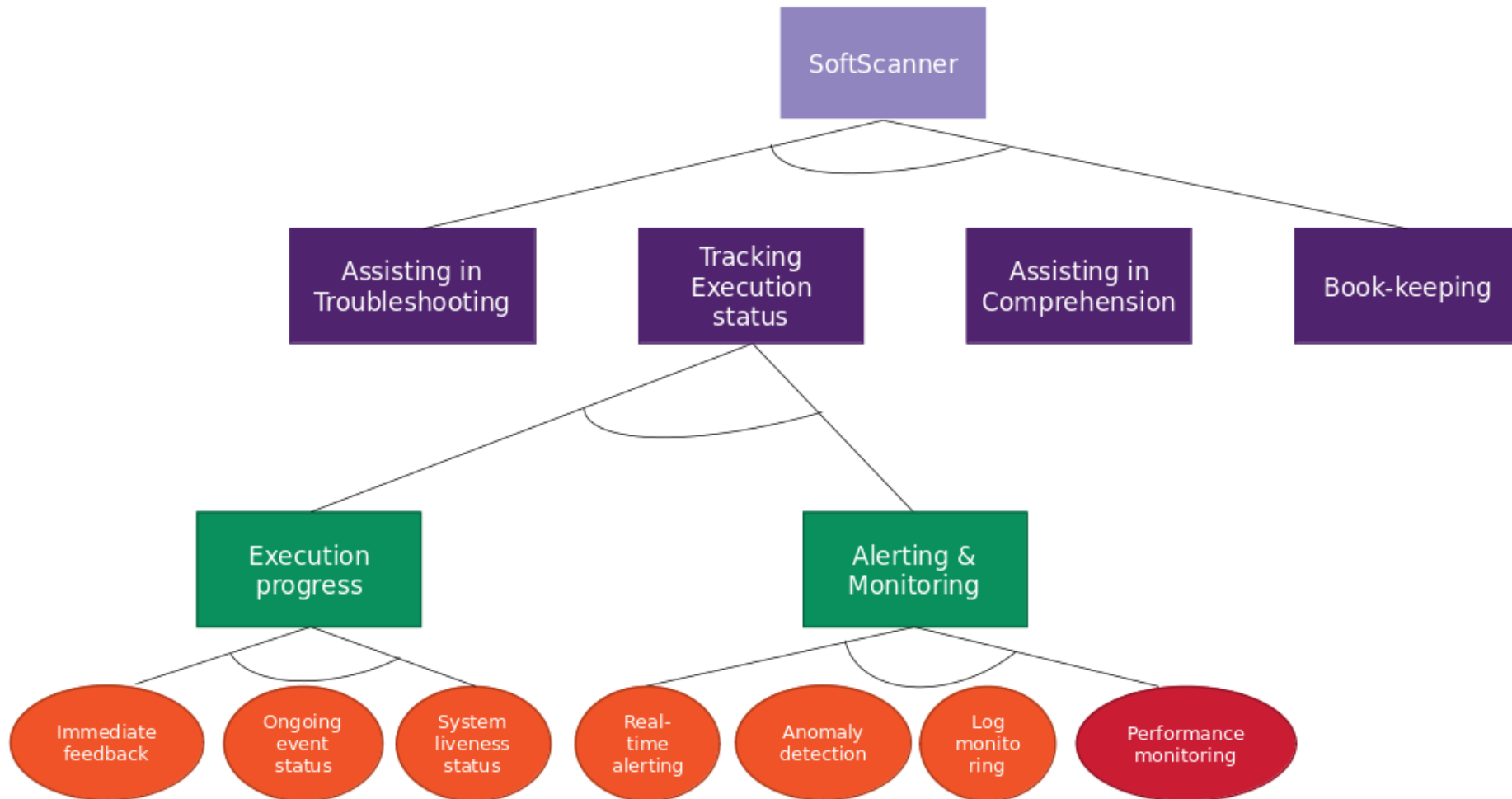
- **Goal:** Allow operators to directly adapt ETCs with respect to the desired tracing goals.
- **Approach:** Leverage/get inspired from existing visualization platforms (such as Kibana dashboard) and integrate/adapt them to the project based on the requirements.



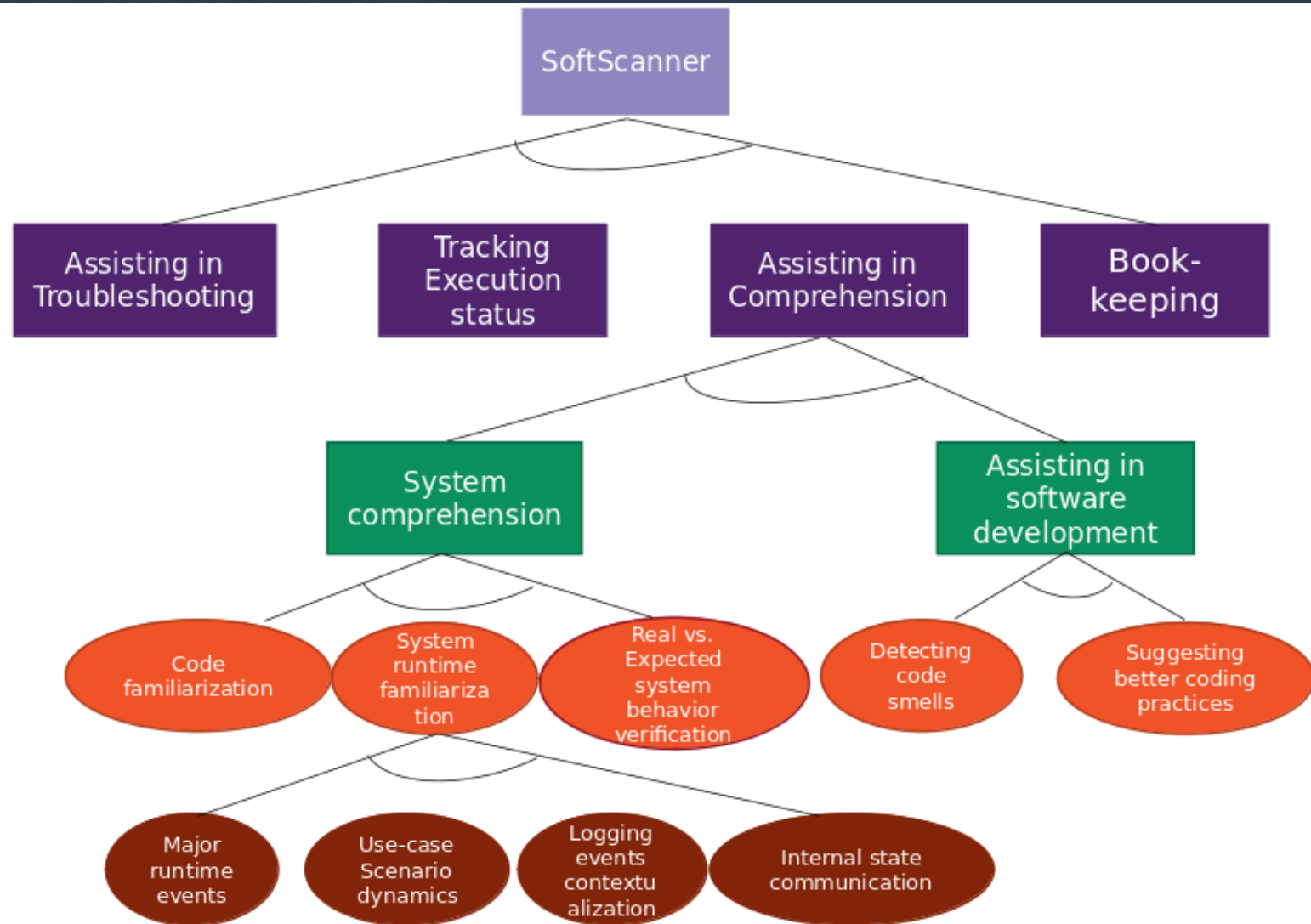
# SoftScanner Feature Model (1)



# SoftScanner Feature Model (2)



# SoftScanner Feature Model (3)



# SoftScanner Feature Model (4)

0

SoftScanner

1

Assisting in  
Troubleshooting

Tracking  
Execution  
status

Assisting in  
Comprehension

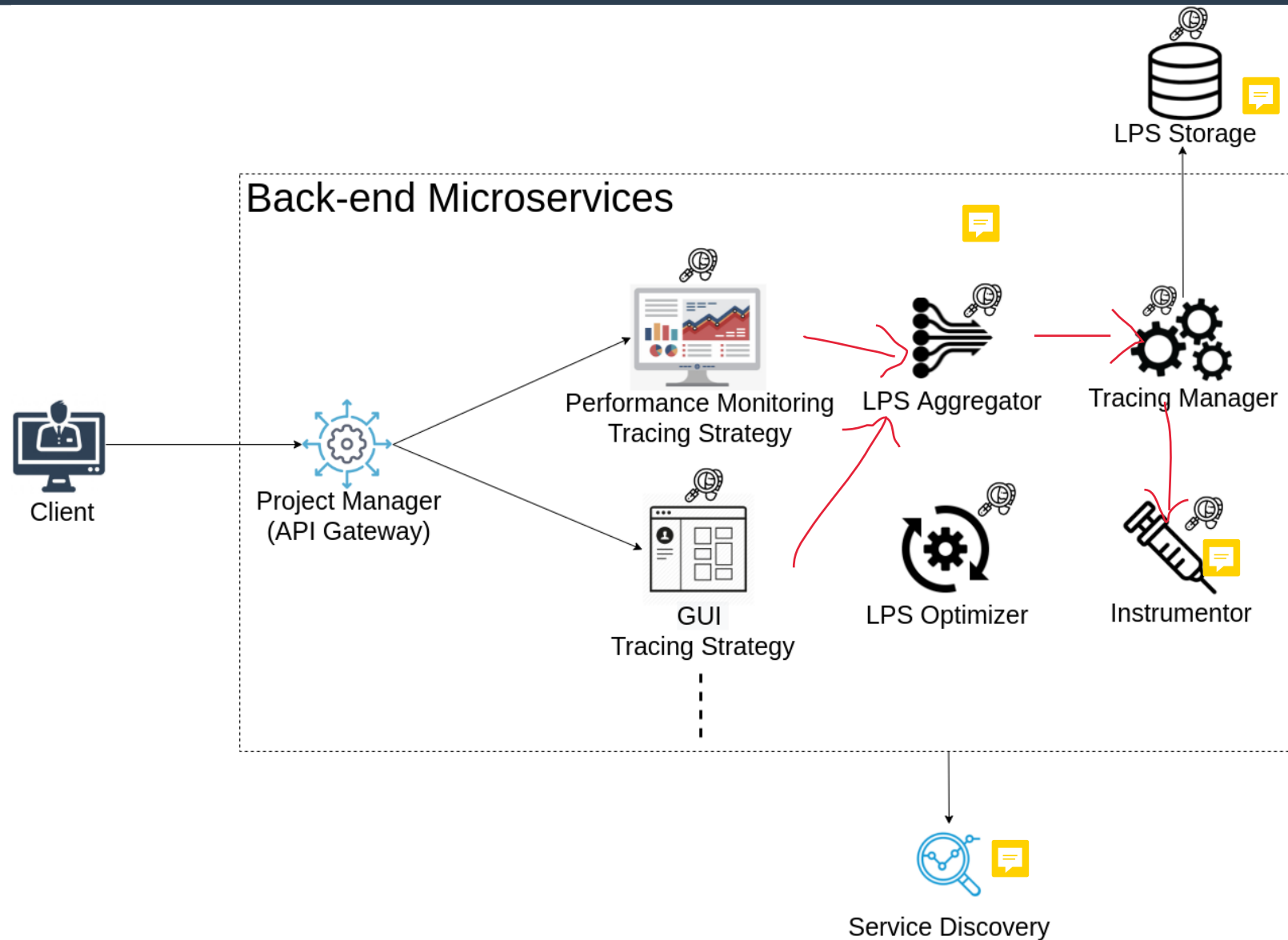
Book-keeping

2

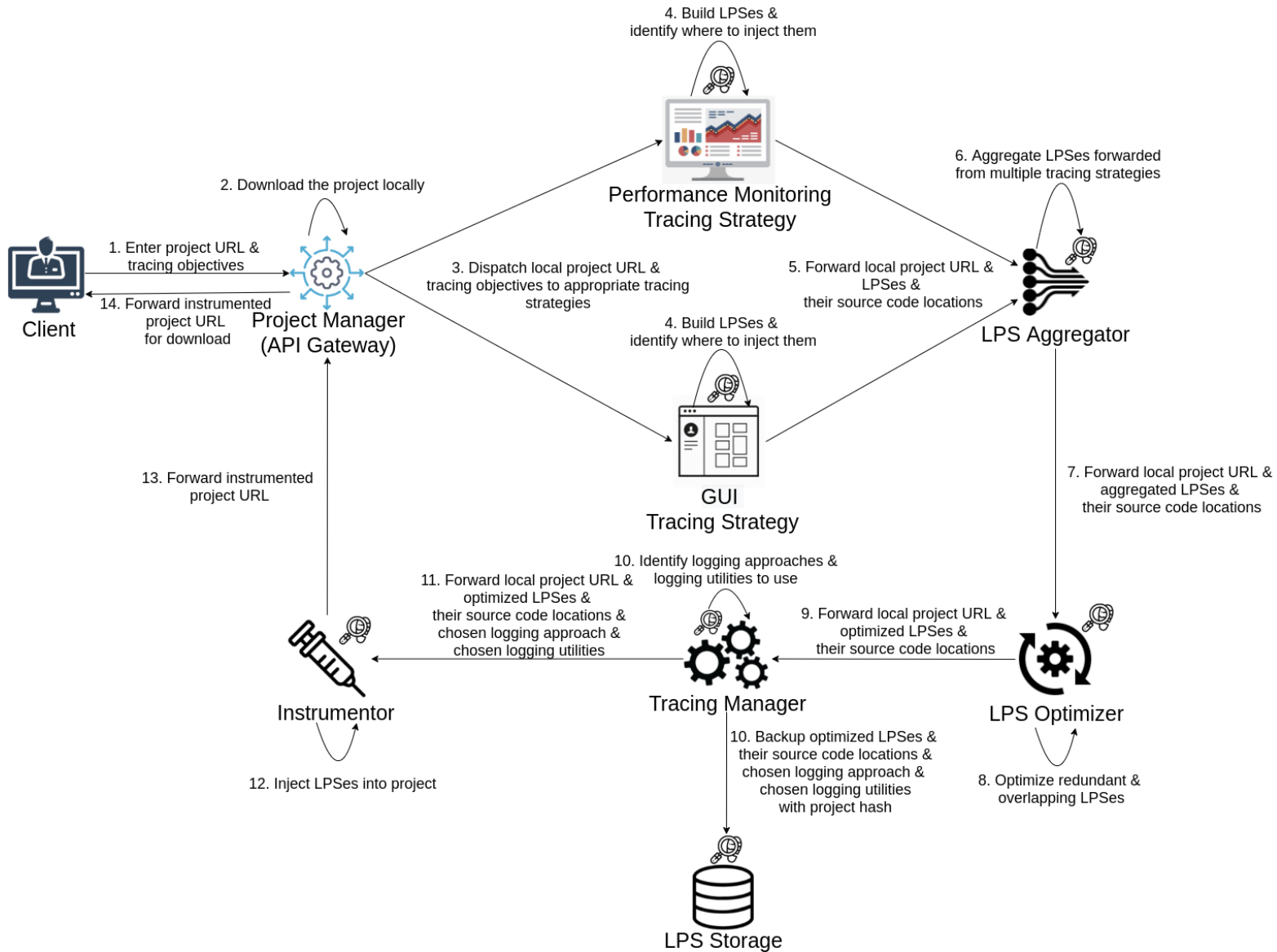
Runtime  
statistics

Auditing

# SoftScanner Architecture

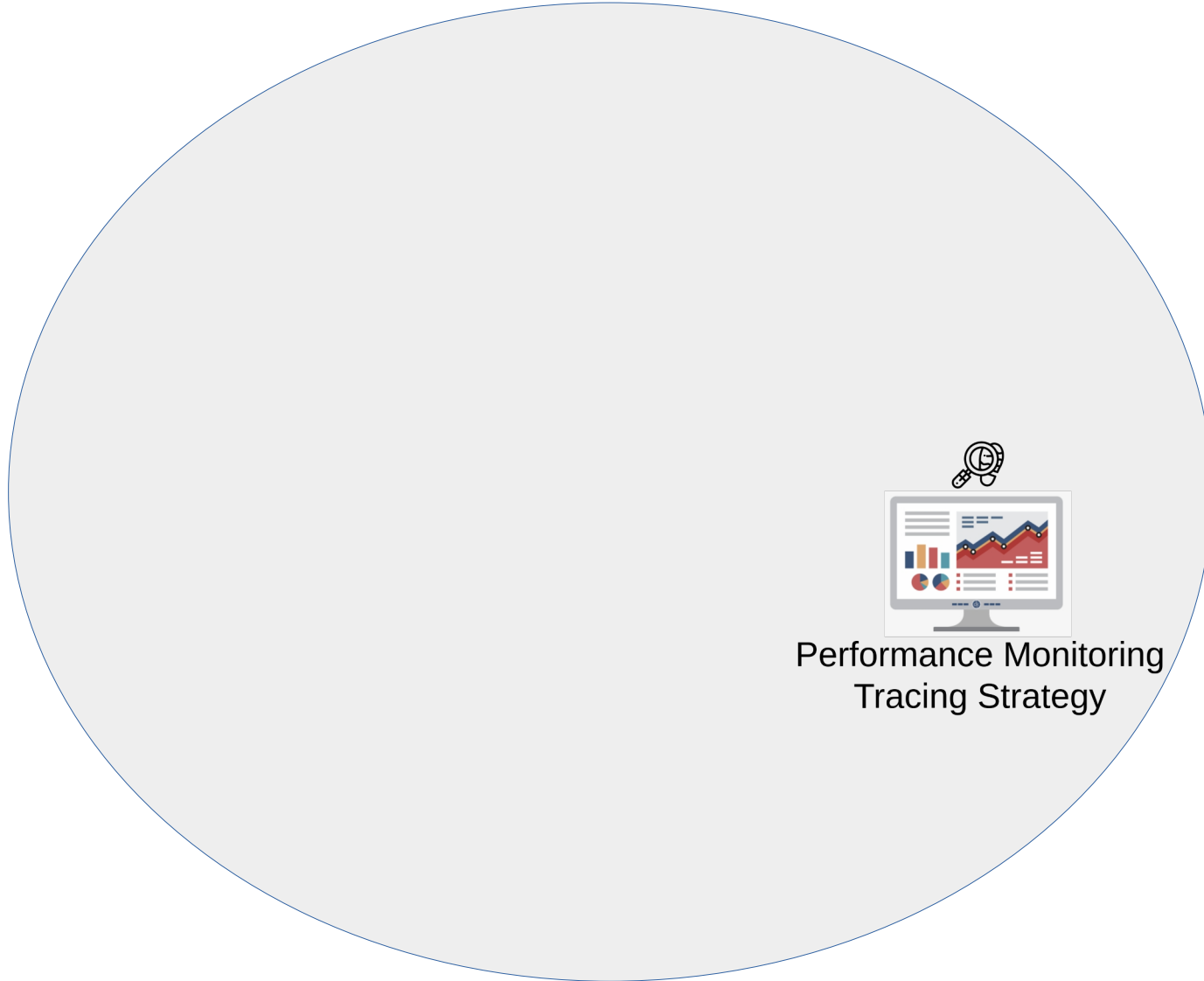


# SoftScanner Workflow






# SoftScanner Ecosystem



# Performance Monitoring Tracing Strategy (1)

- Based on: Log4Perf (Yao et al., 2018).
- Goal: identify performance-influencing programming entities of different granularities (components, methods, basic blocks).
- Methodology:
  - Run execution scenarios on a project
  - Build statistical performance models based on existing logs and logs injected at the beginning of every iteration for each granularity
  - Inject LPS for each performance-influencing programming entity.
- Performance Metric: Mean CPU usage percentage

Performance-Influencing Granularity Level	# Logs Generated During Simulation	# Performance-Influencing Entities	Mean CPU Usage Percentage 
Components (Web Requests)	13138	6	64.95%
Methods	5863	1	65.67%
Basic blocks	5169	0	70.91%

# Performance Monitoring Tracing Strategy (2)

Example of performance-influencing components

```
[{"granularityType":"web request","granularityId":"InputComponent"},  
 {"granularityType":"web request","granularityId":"ColorPipe"},  
 {"granularityType":"web request","granularityId":"SelectPerson"},  
 {"granularityType":"web request","granularityId":"ResearchPersons"},  
 {"granularityType":"web request","granularityId":"DeletePerson"},  
 {"granularityType":"web request","granularityId":"EditeurTagsComponent"}]
```

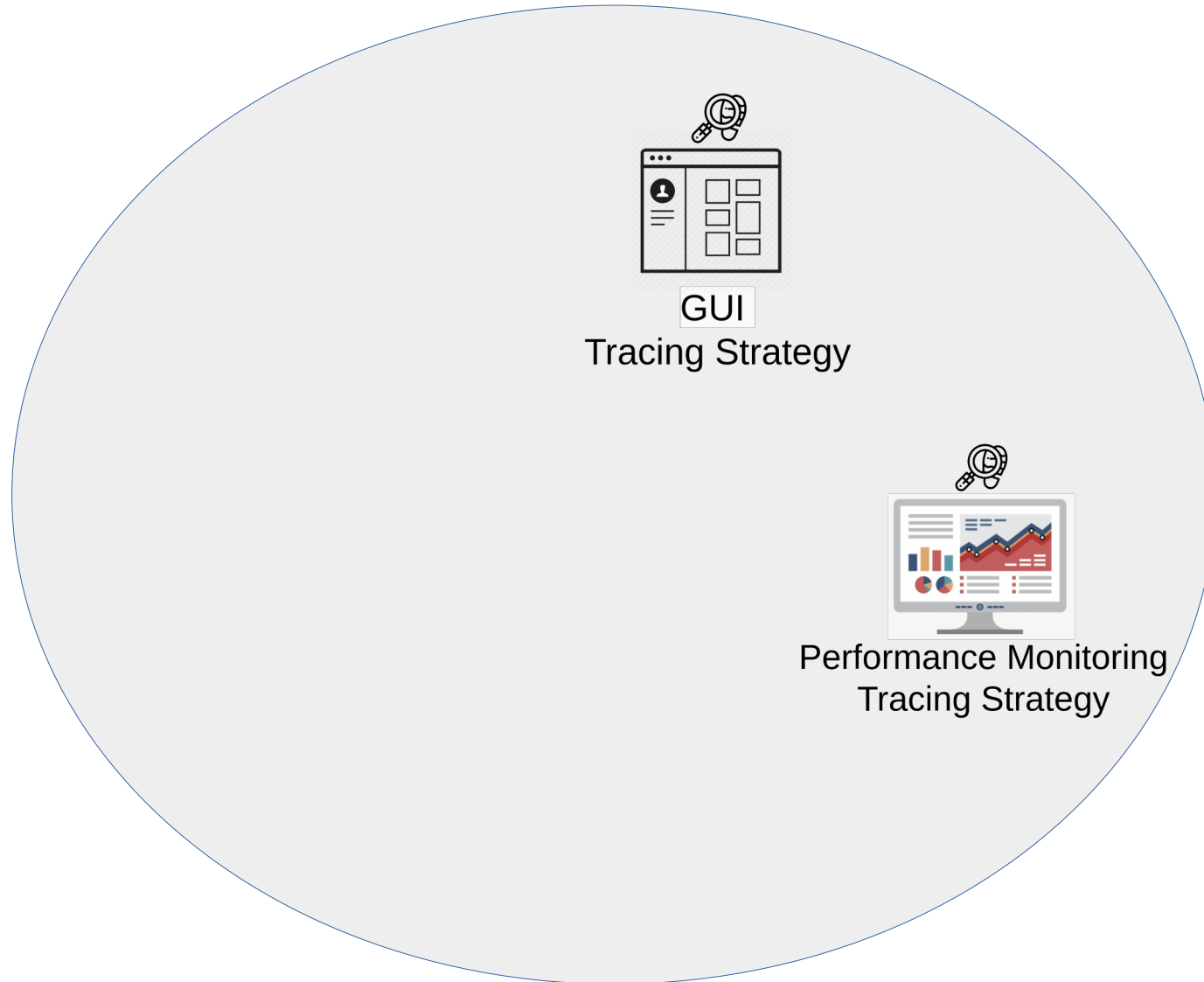
Example of performance-influencing methods

```
[{"granularityType":"method","granularityId":"constructor(public research:string)#ResearchPersons"}]
```

Example of performance-influencing basic blocks

```
[{"granularityType":"web request","granularityId":"InputComponent"},  
 {"granularityType":"web request","granularityId":"ColorPipe"},  
 {"granularityType":"web request","granularityId":"SelectPerson"},  
 {"granularityType":"web request","granularityId":"DeletePerson"},  
 {"granularityType":"web request","granularityId":"EditeurTagsComponent"},  
 {"granularityType":"method","granularityId":"constructor(public research:string)#ResearchPersons"}]
```

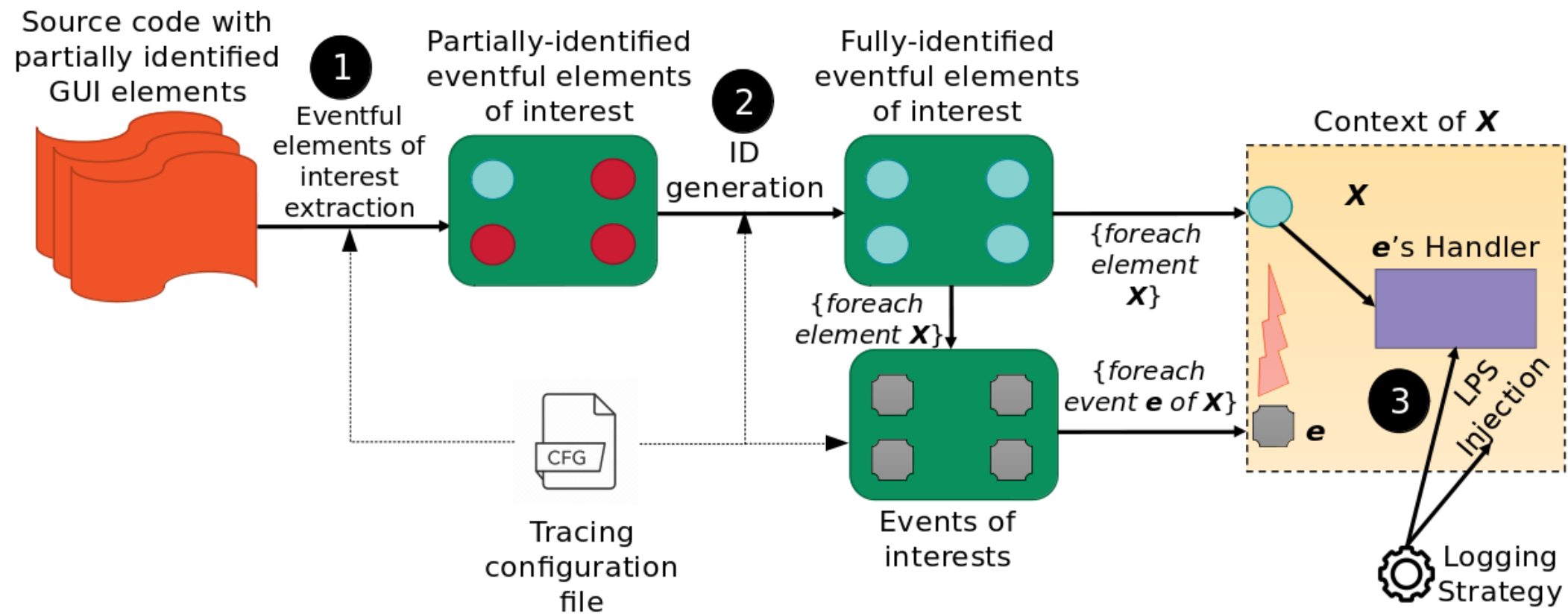
# SoftScanner Ecosystem – continued



# GUI Tracing Strategy (1)

- Goal: provide a tracing mechanism for GUI widgets.
- Methodology: given a set of target events, trace all widgets upon which these events can be triggered.
- Currently supported events:
  - **Click:** clickable widgets.
  - **Submit:** form submissions. (*ngSubmit for Angular*)
  - **Href:** hyperlinkable widgets. (+ *routerLink for Angular*)
  - **FocusOut:** meaningfully focusable widgets (e.g., *<input> widgets like text fields, textareas, etc.*)
  - **Change:** meaningfully changeable widgets (e.g., *drop-down select lists, checkboxes, radio buttons, etc.*)
  - **File:** file uploads.

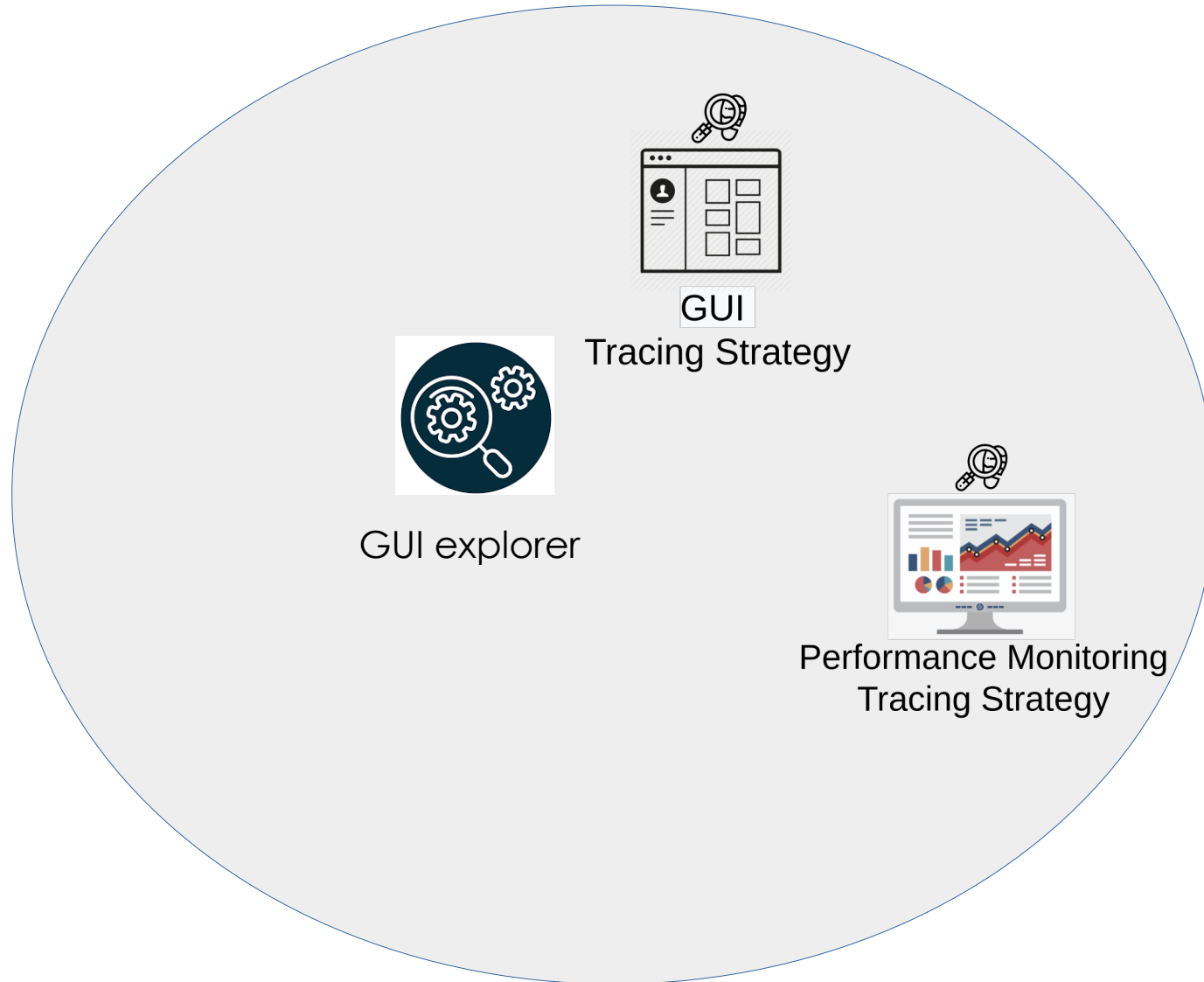
# GUI Tracing Strategy (2)



# GUI Tracing Strategy (3)



# SoftScanner Ecosystem – continued



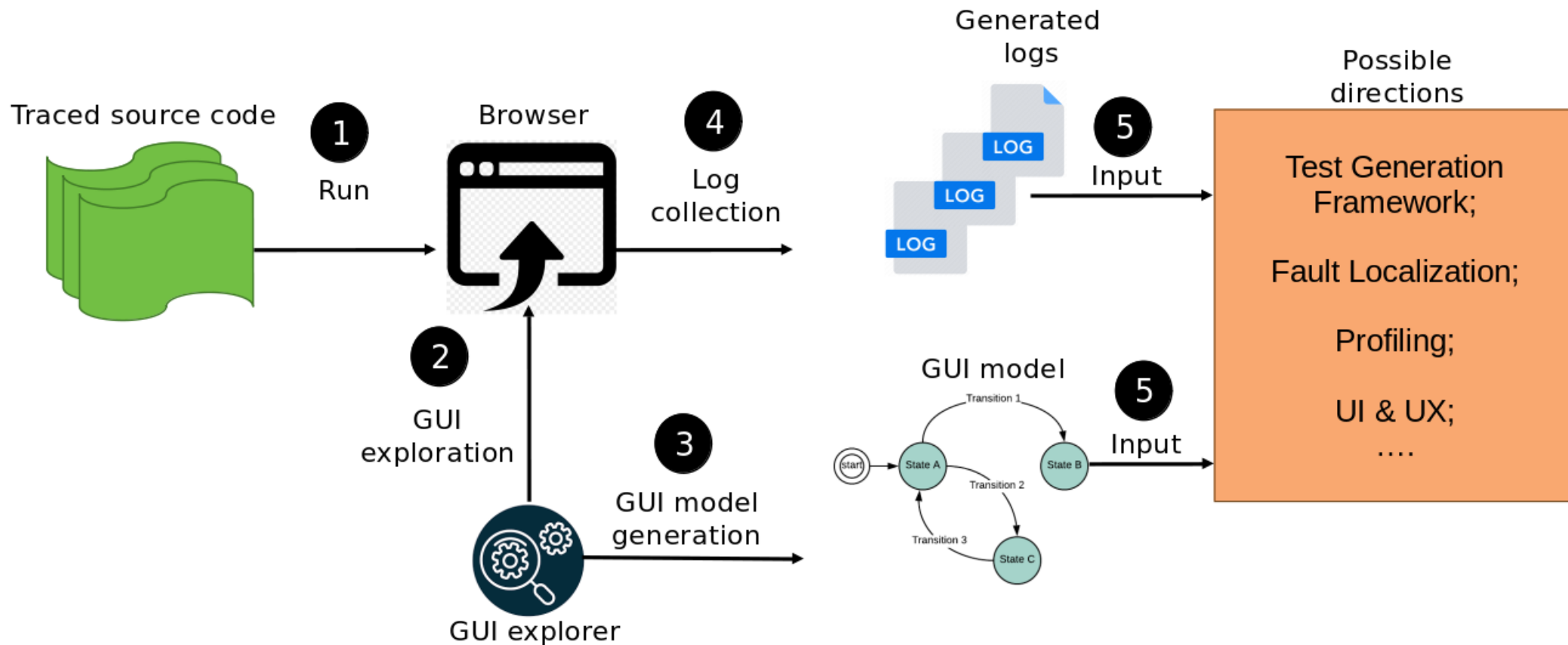


# GUI Exploration (1)

*(Work in progress)*

- Based on:  RIPuppet (J. D. Osorio, 2019) and  GUI Ripping (A. Memon et al., 2003)
- Goal: Model a web application's GUI and leverage it for different software engineering tasks, including tracing, a framework for test generation, profiling, UI & UX, ...
- Methodology:
  - Construct a GUI state machine consisting of GUI states and transitions using Selenium.
  - **GUI state:** a set of widgets and **executable widgets** (i.e., *GUI widgets tagged with an executable heuristic and interacted with by Selenium*).
  - **GUI state transition:** a source state, a target state, the event causing the transition, and the source state widget where the event has been triggered.

# GUI Exploration (2)



# GUI Exploration (3)



# Current State of Affairs (1)

Service/Theme	Status	Notes
Performance Monitoring Tracing	[check]	Optimisation Phase
GUI Tracing	[check]	Optimisation Phase
Instrumentor	[check]	Optimisation Phase
Project Manager	[check]	1. More test apps required 2. Feature addition phase
GUI Explorer	[check]	1. More test apps required 2. Feature addition phase
LPS Aggregator	[pending]	-
LPS Optimizer	[pending]	-
Tracing Strategy Manager	[pending]	-
Test Generation Framework	[pending]	More in depth technical analysis required

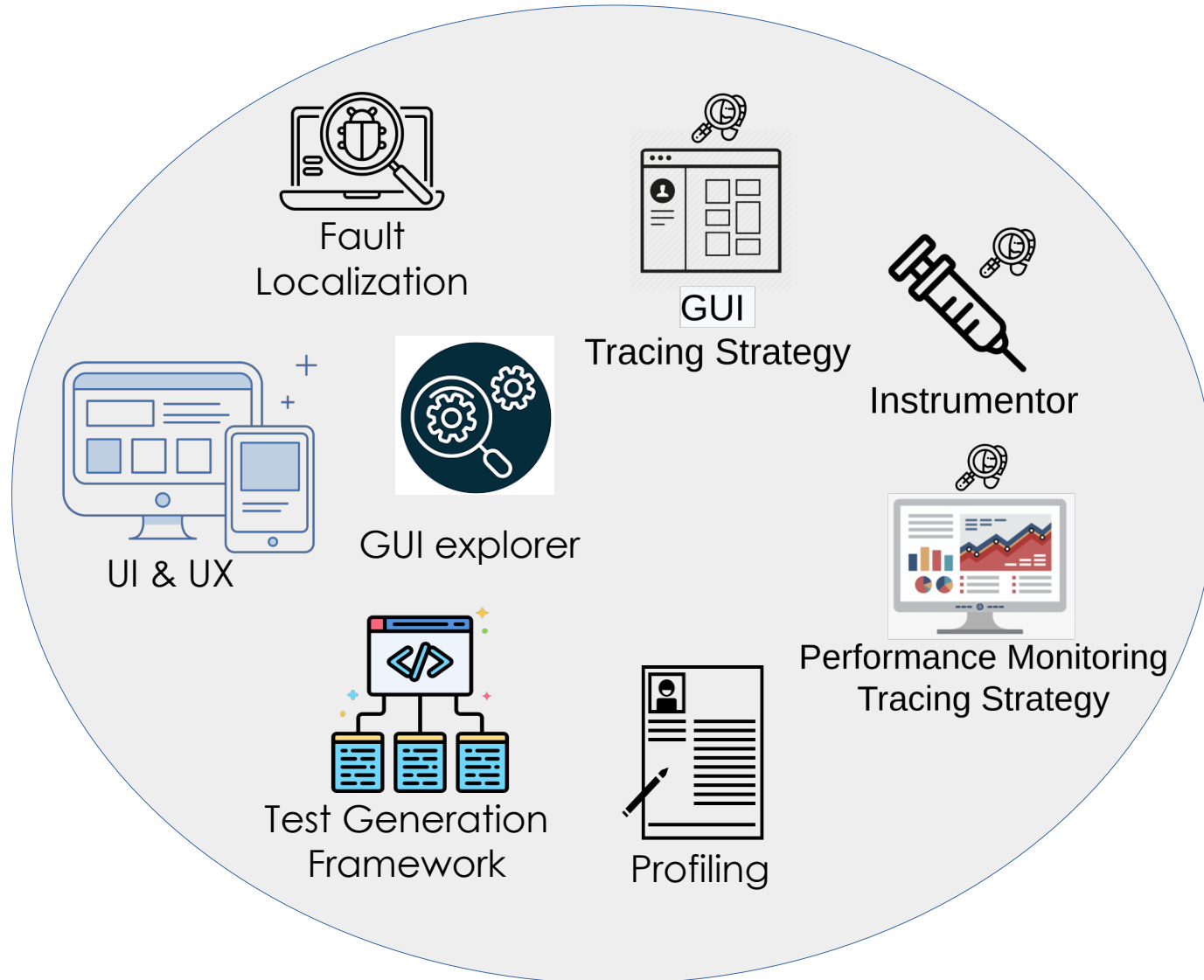
## Current State of Affairs (2)

Service/Theme	Status	Notes
Profiling	[check]	More research required
UI & UX	[pause]	More research required
Fault Localization	[pause]	More research required
Security Tracing	[pause]	A lot of research required

# Future Works

- **Tracing Manager**
  - Supporting Multiple Logging Approaches
  - Integrating and Managing LUs
- **LPS Optimizer**
  - Overlapping ETCs Optimizations
  - Redundant ETCs Optimizations
- **Performance Monitoring Logging Strategy**
  - Baseline LPS injection for projects with no readily-available LPSes.
  - Usage of the RAIL performance model with universal/custom performance metrics.
- **GUI Exploration**
  - Test on more complex applications (*waiting for BLHub*)
  - Semi-automated form filling using a JSON configuration file for problematic input fields (e.g., sign-in, sign-out, password-requiring views, etc.)
  - Pop-ups
  - Use conjointly with generated GUI widget traces for profiling, usage analysis, etc
  - Graph-oriented databases.

# SoftScanner Ecosystem – To be continued...

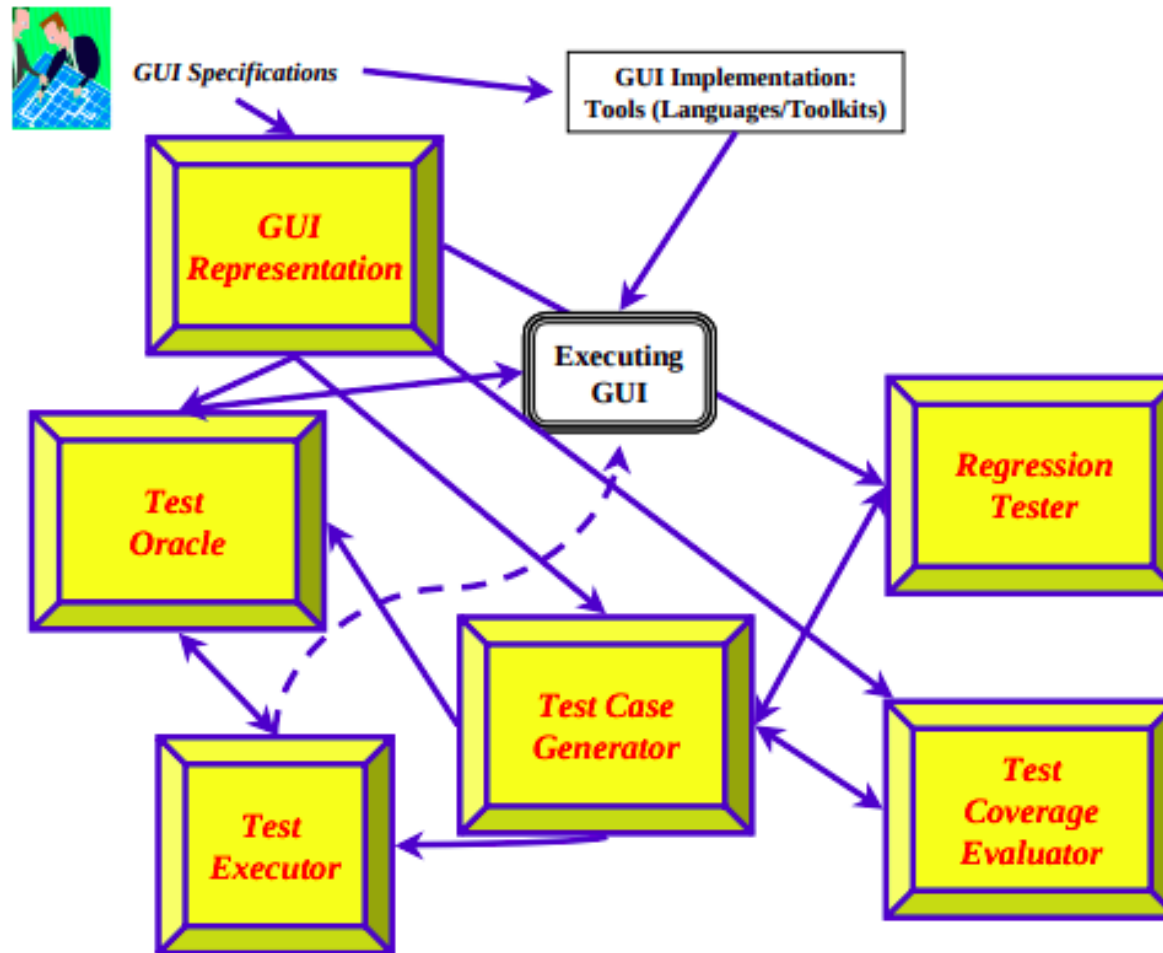


# Extras



# Test Generation Framework

*(Work in progress)*



(A. Memon, 2001)

# Logging Approaches (1)

## Conventional Logging

- Import and use a Logging Utility (LU), where generally a logging object is created and used to inject the LPSes.
- The generated log messages are usually loosely formatted in natural language and cannot be easily parsed by computer programs.
- Very easy to setup and the LPSes can be placed almost anywhere in the SUS (System Under Study).
- The LPSes are scattered across the entire system and tangled with the feature code → maintenance and evolution more challenging as the SUS evolves.

# Logging Approaches (2)

## Rule-based Logging



- Resolves the scattered logging code problem introduced by conventional logging.
- Generalizes the logging behavior by specifying a set of rules.
- Example: with Aspect-Oriented Programming (AOP)-based logging, developers define rules through aspect files, typically consisting of pointcuts (*where to log*) and advices (*what to log*) → LPS modularity improvement.
- Developers only need to revise the rules without modifying code at multiple locations → easier maintenance and evolution of LPSes.
- Lacks flexibility, as it is difficult to generalize individual logging concerns into rules.

# Logging Approaches (3)

## Distributed Tracing

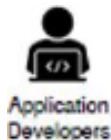
- Motivation: it is more challenging to correlate log messages from different processes/machines in large-scale distributed systems.
- Resolves the LPSes' lack of execution context in conventional and rule-based logging approaches.
- The generated log messages are structured, can be connected by a set of common variables, and are collectively referred to as an end-to-end trace, covering a complete request workflow.
- LUs are developed by library developers which are mainly responsible for the instrumentation task.
- The SUS' developers only need to import the tracing library and perform some setup actions, and perform additional log instrumentations if needed.

# Logging Approaches (4)

Dimension/Approach	Conventional Logging	Rule-Based Logging	Distributed Tracing
Instrumentation	SUS developers	SUS developers	 LU developers
Log Filtering	Verbosity Level	Verbosity Level	Sampling 
Log Format	Free Form	Free Form	Structured
Domain	General	General	Distributed Systems
Flexibility	High	Low	Medium
Log Scatter	High	Low	Low

# Logging Approaches (5)

■ Logging Object 
 ■ Verbosity Level 
 ■ Static Texts 
 ■ Dynamic Content



Application Developers

```

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

*MyServer.java*

(a) Using conventional logging for log instrumentation



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*

```

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

*MyServer.java*

(b) Using rule-based logging for log instrumentation

```

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
...

```

(d) Outputted logs

```


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
...

```

(e) Outputted logs


# Logging Approaches (6)

(c) Use distributed tracing for log instrumentation

 Application Developers

```
1 import io.opentracing.*
2 class MyServer {
3     Tracer tracer = GlobalTracer.get();
4     private start() {
5         Server server = new TracedServer(tracer);
6     }
7     ...
8     void authentication(Request req, ...) {
9         // actual authentication process ...
10        reply(response, ...)
11    }
12 }
```

*MyServer.java*

 Library Developers

```
1 import io.opentracing.*
2 class TracedServer extends Server {
3     ...
4     @Override
5     public void onReceive() {
6         SpanContext parentSpan = tracer.extract(HTTP_HEADERS, headers);
7         spanBuilder = spanBuilder.asChildOf(parentSpan);
8         span = spanBuilder.start();
9         span.log(Map.put("Client", req.userName)
10                    .put("Message", "Receive from client"))
11     }
12     ...
13 }
14 @Override
15 public void onSend(Response response) {
16     ...
17     span.log(Map.put("IP", req.IP)
18                .put("Message", "Send response"));
19     ...
20 }
```

*TracedServer.java*

 Tracing Object  Key  Value

(f) Outputted traces

```
"data" : [
  {
    "traceID": "1242029787ec9011"
    "spans": [
      ...
      {
        "traceID": "1242029787ec9011",
        "spanID": "1a481c39c9e66ac6",
        "parentSpanID": "c53ac490f028963a",
        "duration": 277146,
        ...
        "logs":[
          {
            "timestamp": 1585844561219000,
            "Client": "bob",
            "Message": "Receive from client"
          },
          {
            "timestamp": 1585844561230000,
            "IP": "192.168.0.1",
            "Message": "Send Response"
          },
          ...
        ],
      }
    ],
  },
  ...
]
```

**Thank you for your attention!**  
**Any questions?**

