Exception monitoring for the Flask Monitoring Dashboard

BSc project

The Flask Monitoring Dashboard (FMD) is a valuable tool for analysing Flask application performance. However, prior to this project it lacked built-in support for tracking exceptions, requiring developers to rely on external services or manual log inspection. This project extends the FMD with integrated exception monitoring, designed to require no additional configuration from the user and preserve the FMD's minimal setup philosophy. We designed a new data model to capture both caught and uncaught exceptions, linking them to specific HTTP requests. In the dashboard, we present the exceptions grouped using stack traces and executed function code to help developers detect recurring issues. The result is a unified dashboard that integrates performance and exception monitoring, giving FMD users deeper insight into their applications.

Group member	Email
Albert Juel Ross	alrj@itu.dk
Carmen Alberte Nielsen	alcn@itu.dk
Natalie Clara Petersen	natp@itu.dk

Submission: May 15, 2025

IT University of Copenhagen

BIBAPRO1PE

Contents

1 Acknowledgement

We would like to express our gratitude to our supervisor, Mircea Lungu, for his continuous support and invaluable feedback throughout the course of this project. Through our weekly meetings and many constructive discussions, he helped us determine what to implement and the best approach to achieve it. His genuine interest in the project was both motivating and helpful, and we truly appreciate his dedication.

2 Introduction

In modern web development, ensuring high reliability and availability is essential. Exception monitoring plays a crucial role in achieving this by helping developers identify and understand issues that occur during runtime. When exceptions occur during runtime, they often point to bugs, unexpected inputs, or unhandled edge cases that require attention. As a web application continues to grow in size, it becomes harder to keep track of potential issues, and without proper tracking, these issues can go unnoticed, making debugging and maintenance significantly harder.

One popular framework for web development is Flask: a lightweight Python web framework known for its minimalism and flexibility. Flask provides just the core tools needed to build web applications, making it easy to get started while still supporting the development of large-scale systems[1]. A wide range of extensions also offer additional functionalities. This balance of simplicity and extensibility has made Flask widely adopted in both small projects and systems at large companies[2].

To aid in the process of creating robust and fast Flask web applications, the open source project Flask Monitoring Dashboard (FMD) was created, first released in 2017 in conjunction with a short paper written by Vogel Et al. [Vog+17]. The FMD offers several advantages compared to other monitoring services: it is free of subscriptions or fees, self-hosted, ensuring control of the data, and requires minimal setup, needing only a single line of code in the main application. Prior to this project, the FMD did not contain any exception monitoring, which required users to search through logs or use external tools for debugging.

Integrating an exception module into the FMD system would offer a unified solution, allowing developers to monitor both performance and exceptions in a single dashboard.

Exception tracking naturally complements performance monitoring by helping to interpret performance data. While exceptions do not always affect performance, they often coincide with failed requests or unusual behaviour and can reveal underlying issues that affect the performance metrics. Combined, performance and exception metrics provide a more complete view of application behaviour, and with the exception module extension, FMD moves closer to being an all-in-one solution for Flask monitoring.

[1]Flask documentation. https://flask.palletsprojects.com/en/stable/# (accessed: 13.05.2025)
[2]Famida. *Top Big Companies Using Python Flask.* https://entri.app/blog/top-big-companies-using-python-flask (accessed: 13.05.2025)

A key focus of the FMD, and in extension this thesis, is to provide functionality with minimal configuration, i.e. make it as simple as possible for the user to take advantage of the functionality. The purpose of this thesis is to extend the FMD with exception monitoring functionality while preserving FMD's core advantages. This leads us to the research question:

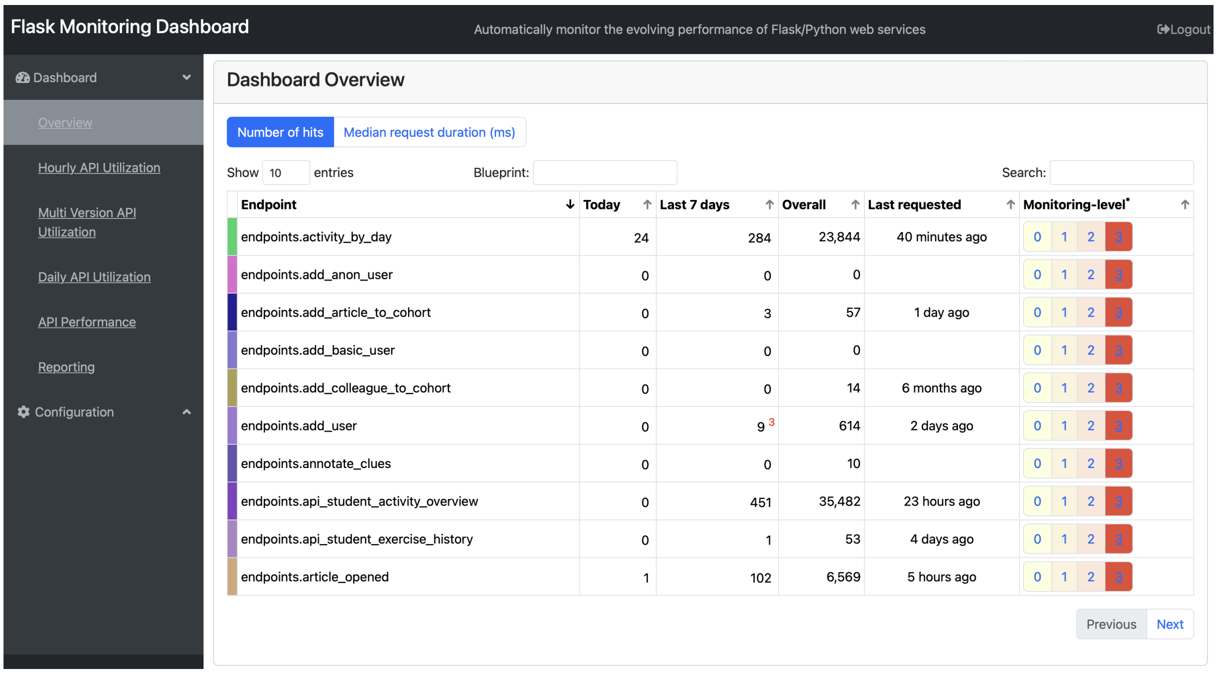
> *RQ*: How can the Flask Monitoring Dashboard be effectively extended to incorporate exception monitoring without user configuration, providing a unified self-hosted solution for Flask monitoring?

In this thesis we will start by providing some background to the problem in Section 3, then we will describe our solution to the research question *RQ* in the design and implementation, Section 5. After describing the design, we will discuss our choices and problems we faced when designing and implementing in Section 6, and at last moving to the concluding remarks in Section 7.

3 Background

3.1 FMD

The FMD is an open-source tool designed to monitor the performance of Flask web applications and require minimal configuration. It provides information on runtime metrics, such as request counts for different endpoints and the duration of function execution[3].



Endpoint	Today	Last 7 days	Overall	Last requested	Monitoring-level*
endpoints.activity_by_day	24	284	23,844	40 minutes ago	0 1 2 3
endpoints.add_anon_user	0	0	0		0 1 2 3
endpoints.add_article_to_cohort	0	3	57	1 day ago	0 1 2 3
endpoints.add_basic_user	0	0	0		0 1 2 3
endpoints.add_colleague_to_cohort	0	0	14	6 months ago	0 1 2 3
endpoints.add_user	0	9 3	614	2 days ago	0 1 2 3
endpoints.annotate_clues	0	0	10		0 1 2 3
endpoints.api_student_activity_overview	0	451	35,482	23 hours ago	0 1 2 3
endpoints.api_student_exercise_history	0	1	53	4 days ago	0 1 2 3
endpoints.article_opened	1	102	6,569	5 hours ago	0 1 2 3

Figure 1: Example of Flask Monitoring Dashboard, before this project

Since not all endpoints are equally relevant for monitoring, the FMD offers four monitoring levels: 0, 1, 2, and 3. Monitoring level 0 does not store any data in the database, it

[3] Flask-MonitoringDashboard. https://github.com/flask-dashboard/Flask-MonitoringDashboard (accessed: 24.03.2025)

only updates a global timestamp for the latest request. Level 1, 2, and 3 store increasing amounts of performance data and track status codes.

Prior to this project FMD provided no information about exceptions. For failed API requests, it only recorded the number of failures (see red number in Figure 7), their status codes, and timestamps.

3.2 Distinguishing Exceptions

In the context of flask web applications, we categorize Exceptions into two groups:

- Uncaught exceptions - These occur when an error is not caught within the application logic, leading to a crash and an automatic HTTP 500 status code returned by flask.
- Caught exceptions - These are anticipated errors that are caught and managed within the code using try-except blocks or similar mechanisms. As developers might want to log caught exceptions, these are also relevant in our project.

In a Flask app, a request represents a single HTTP interaction from a client. Multiple caught exceptions can occur within a single request, as errors may be caught and managed at different stages of execution. However, at most one uncaught exception can occur per request, as it will immediately terminate execution.

3.3 Stack Traces in Application Logs

The stack traces that appear in the application logs (See Figure 2), provide both the line number, function name, and file path of the executed code in each frame. Thereby, they point to the exact locations in the code where the exceptions occurred.

```
Traceback (most recent call last):
  ...
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
      flask_monitoringdashboard/main.py", line 106, in c
    return b()
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
      flask_monitoringdashboard/main.py", line 102, in b
    return a()
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
      flask_monitoringdashboard/main.py", line 98, in a
    raise Exception("ohhh no!")
Exception: ohhh no!
```

Figure 2: Simple stacktrace logged by python [4]

[4]For a full example see Appendix A

3.4 Stack Traces in Python

In Python, stack trace information is stored in an exception object's `__traceback__`[5] attribute which has the type `TracebackType` (or `None`).

```python
class BaseException:                 # Common base class for all exceptions
    __traceback__: TracebackType | None
    ...

class TracebackType:
    tb_next: TracebackType | None # Next TracebackType in the chain
    tb_frame: FrameType           # Execution frame
    tb_lineno: int                # Line that invokes the next function
                                  # or raises the exception
    ...

class FrameType:
    f_code: CodeType           # The CodeType object executed in this frame
    ...

class CodeType:
    co_name: str               # Function name
    co_filename: str           # File name
    co_firstlineno: int        # First line number of the function
    ...
    # use native inspect.getsource() on this object to get the
    # function code as string
```

Figure 3: TracebackType from the Python standard library, showing the fields we utilize.

The `__traceback__` field is a linked list, representing the sequence of function calls that led to the exception. Each node in this list is a `TracebackType` object, representing a single stack line. Collectively, these objects form the complete stack trace. Each `TracebackType` object points to a `FrameType` object that contains metadata on the stack frame. Thereby `TracebackType` acts as a wrapper around the stack frame for the purpose of modelling a stack trace. Some of the relevant metadata includes, the line where the exception occurred, the name and source code of the executed function, and the file in which it is defined (see Figure 3).
While only the last `TracebackType` represents where the exception was raised, the others trace the sequence of function calls that led to it. The `tb_next` attribute gives the ability to step through[6] the stack trace.

[5]Python docs on TracebackType. https://docs.python.org/3/reference/datamodel.html#traceback-objects (accessed: 24.03.2025)
[6]"Stepping through" in this context means going from the current stack frame to the next, starting from the invocation point and continuing until where the exception was raised.

4 Related work

Flask users might rely on user feedback or manual inspection of application logs to diagnose an issue [Che19]. However, these approaches can be fragmented and lack structure. They provide limited support for identifying recurring problems or understanding the full scope of an issue over time[7]. Therefore, users might seek insights through external monitoring tools, and there exist many such frameworks compatible with Flask applications.

One of the most widely used tools for capturing and analysing exceptions in web applications is Sentry. Sentry captures exceptions and provides users with detailed information regarding these. Sentry helps developers by grouping similar exceptions into issues, resulting in a clear overview that makes recurring problems easier to detect[8]. In addition, it also provides performance monitoring.

Another popular tool for exception logging is Bugsnag, which offers many of the same capabilities as Sentry for Flask projects. The differences in this context lie mainly in API design and configuration workflows.

Using either Sentry or Bugsnag might seem like an obvious choice, as they address almost all imaginable needs, but for users who only seek the most basic functionalities of monitoring, the variety of features and customization they provide might overwhelm unnecessarily.

Both Bugsnag and Sentry, offer free plans for individual developers, but charge for team usage[9], making them less accessible for collaborative projects. The free plans restricts the range of features available, and in the case of Bugsnag, set a fixed monthly limit on the number of errors recorded. The fact that they are hosted externally might raise security concerns, as the user will send their data to external parties. Although there are ways of hosting Sentry yourself[10], doing so can be quite difficult and resource intensive; Sentry requires a lot of resources to run, with a minimum of 16 gigabytes of ram and 4 cpu cores.[11]

In contrast to services such as Sentry and Bugsnag, another option is to choose a smaller self-hosted alternative, such as Error Tracker[12]. Error Tracker is designed for Python applications and offers a “batteries-include” approach to exception monitoring. In addition to logging exceptions, it also logs requests’ metadata such as headers and scrubs sensitive data. It also includes basic notification hooks and exception filtering. However, compared

[7]raygun, *Error monitoring and exception handling in large-scale software projects.* https://raygun.com/blog/errors-and-exceptions (accessed: 11.02.2025)

[8]sentry-docs, *Issue Grouping.* https://docs.sentry.io/concepts/data-management/event-grouping (accessed: 11.02.2025)

[9]Bugsnag pricing: https://www.bugsnag.com/pricing/ and Sentry pricing: https://sentry.io/pricing/ (accessed: 11.04.2025)

[10]Self hosting Sentry. https://develop.sentry.dev/self-hosted/ (accessed: 10.05.2025)

[11]In contrast, the FMD was benchmarked in the thesis ”Improving the Performance of a Performance Monitor. The case of the Flask Monitoring Dashboard” [Pet19] using a MacBook Pro 2015 with dual-core Intel i5 2.7 GHz, 8 GB LPDDR3 (1866 MHz) RAM, SQLite database, and Flask’s development server. No explicit minimum hardware requirements were stated, but since the benchmarking and testing were successfully run on this hardware, it is reasonable to conclude that FMD runs effectively on systems with similar or better specifications.

[12]Error Tracker readme. https://github.com/sonus21/error-tracker?tab=readme-ov-file#introduction (accessed: 07.04.2025)

to tools like Sentry and Bugsnag, Error Tracker does not display the source code that triggered the exception and uses a fairly simple grouping strategy (see Appendix B).

Bugsink is yet another exception tracking service, which differs from Sentry and Bugsnag in the fact that easy self hosting is a prime value proposition[13]. In contrast to Error Tracker, Bugsink does display source code in stack traces as with Sentry and Bugsnag.

Although Bugsink allows for easy self-hosting, it still functions as a separate service that must be installed and configured alongside the main Flask application. Additionally, both Bugsink and Error Tracker lack performance metrics, meaning developers would still need another tool, such as FMD, for performance tracking.

5 Design and Implementation

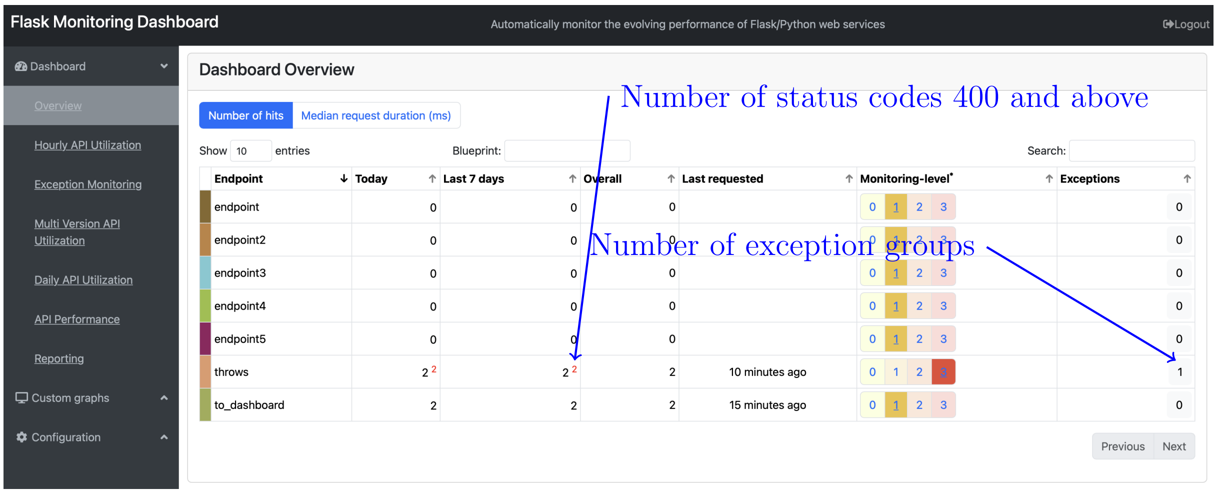
Since the FMD is an existing tool actively used by many parties[14], a key design requirement was to avoid modifying or deleting the existing user data. The primary goal was to introduce new functionality without requiring users to take any manual actions, such as applying migrations or performing data transformations.

5.1 Frontend design

5.1.1 Visual Design

For the frontend UI, we took inspiration from other parts of the FMD with the underlying goal being, that users should be able to easily access information about the latest occurred exceptions.

The main dashboard presents general information about endpoints. In our case, it shows information about how many exceptions have been thrown (see Figure 4).


Flask Monitoring Dashboard
Automatically monitor the evolving performance of Flask/Python web services
Logout
Dashboard
Overview
Hourly API Utilization
Exception Monitoring
Multi Version API Utilization
Daily API Utilization
API Performance
Reporting
Custom graphs
Configuration
Number of status codes 400 and above
Number of exception groups
Previous
Next

Dashboard Overview

Number of hits | Median request duration (ms)

Show 10 entries   Blueprint: Search:

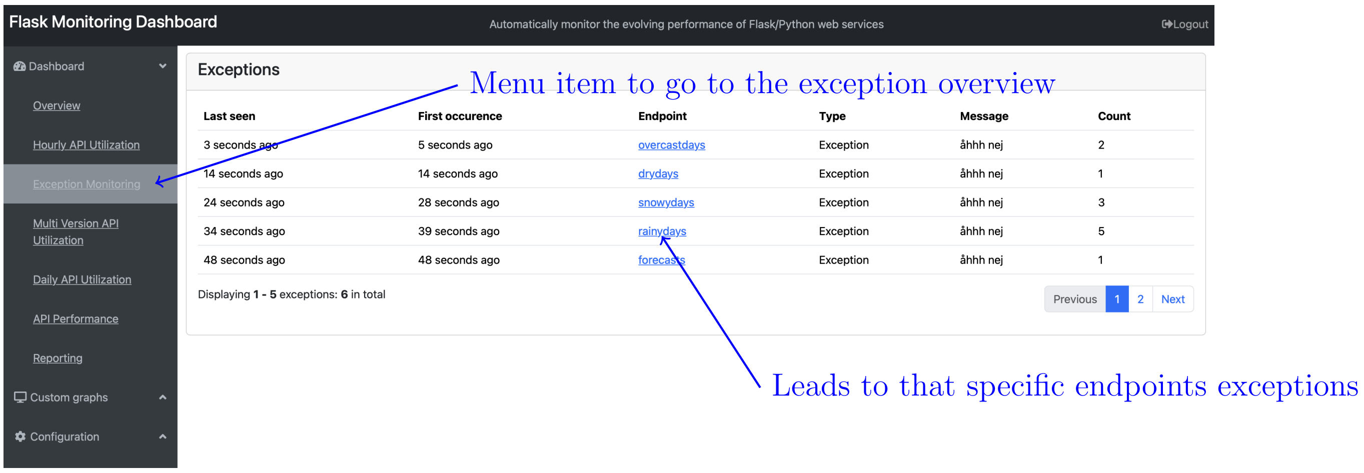
Endpoint	Today	Last 7 days	Overall	Last requested	Monitoring-level*	Exceptions
endpoint	0	0	0		0 1 2 3	0
endpoint2	0	0	0		0 1 2 3	0
endpoint3	0	0	0		0 1 2 3	0
endpoint4	0	0	0		0 1 2 3	0
endpoint5	0	0	0		0 1 2 3	0
throws	2 ^{2}	2 ^{2}	2	10 minutes ago	0 1 2 3	1
to_dashboard	2	2	2	15 minutes ago	0 1 2 3	0

Figure 4: FMD overview page

[13]Self hosting Bugsink. https://www.bugsink.com/built-to-self-host/ (accessed: 10.05.2025)
[14]The flask-monitoringdashboard has more than 500 unique users, https://flask-dashboard.github.io/fmd-telemetry/#/stats/user-session (accessed: 12.05.2025)

It is important to clarify that the red number pointed to in Figure 4 does not represent the number of exceptions directly. It indicates the number of requests that resulted in a non-successful (bad) status code, including requests where an uncaught exception occurred. The monitoring levels decide what data should be collected for a specific endpoint. In our implementation exception data is collected at every monitoring level except level zero. The grey button in the "Exception" column, shows the number of exception groups[15] occurred for that specific endpoint, and when clicked, leads us to the exception overview for that endpoint.

To provide a lightweight overview of all exceptions in the application, users can access the "Exception Monitoring" page via the menu (see Figure 5). This view summarizes the exception groups, with quick navigation to the affected endpoint where the exception can be viewed in more detail.


Flask Monitoring Dashboard
Automatically monitor the evolving performance of Flask/Python web services
Logout
Dashboard
Overview
Hourly API Utilization
Exception Monitoring
Multi Version API Utilization
Daily API Utilization
API Performance
Reporting
Custom graphs
Configuration
Menu item to go to the exception overview
Leads to that specific endpoints exceptions
Previous
1
2
Next

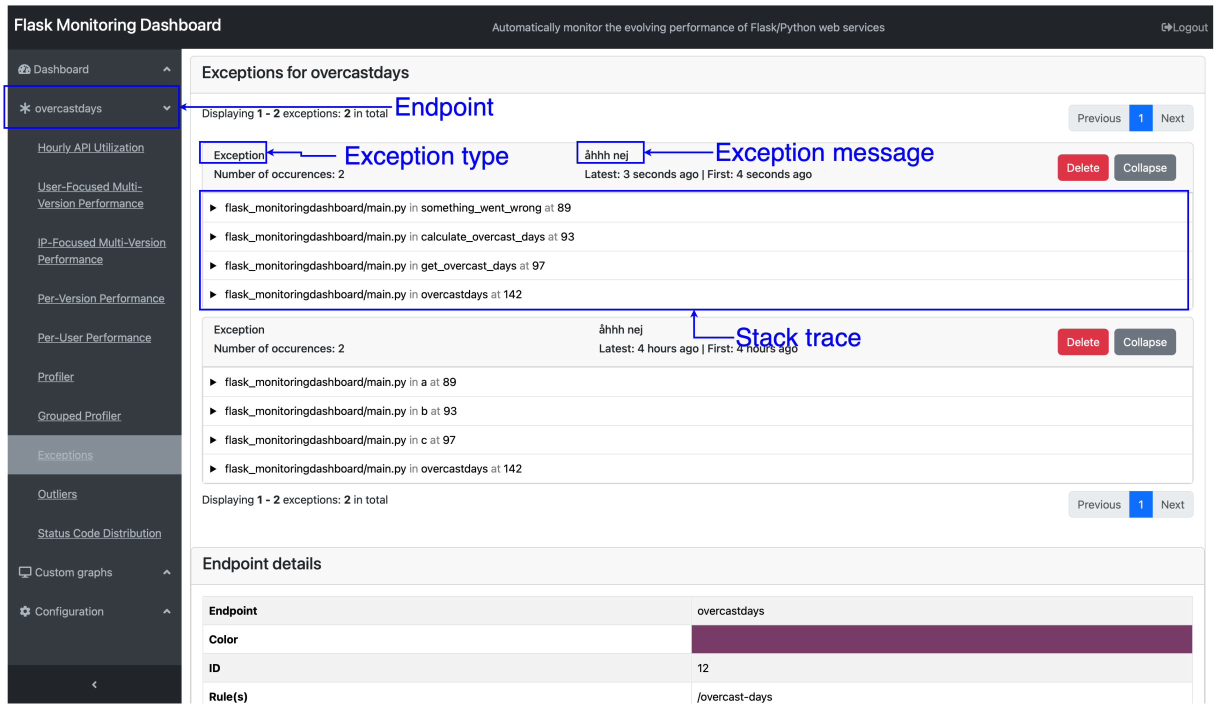
Exceptions

Last seen	First occurence	Endpoint	Type	Message	Count
3 seconds ago	5 seconds ago	overcastdays	Exception	åhhh nej	2
14 seconds ago	14 seconds ago	drydays	Exception	åhhh nej	1
24 seconds ago	28 seconds ago	snowydays	Exception	åhhh nej	3
34 seconds ago	39 seconds ago	rainydays	Exception	åhhh nej	5
48 seconds ago	48 seconds ago	forecasts	Exception	åhhh nej	1

Displaying **1 - 5** exceptions: **6** in total

Figure 5: Exception monitoring overview

[15] An exception group refers to multiple occurrences of exceptions that we treat as one entity.



Exceptions for overcastdays

Displaying **1 - 2** exceptions: **2** in total

Exception — åhhh nej
Number of occurences: 2 — Latest: 3 seconds ago | First: 4 seconds ago

```
▸ flask_monitoringdashboard/main.py in something_went_wrong at 89
▸ flask_monitoringdashboard/main.py in calculate_overcast_days at 93
▸ flask_monitoringdashboard/main.py in get_overcast_days at 97
▸ flask_monitoringdashboard/main.py in overcastdays at 142
```

Exception — åhhh nej
Number of occurences: 2 — Latest: 4 hours ago | First: 4 hours ago

```
▸ flask_monitoringdashboard/main.py in a at 89
▸ flask_monitoringdashboard/main.py in b at 93
▸ flask_monitoringdashboard/main.py in c at 97
▸ flask_monitoringdashboard/main.py in overcastdays at 142
```

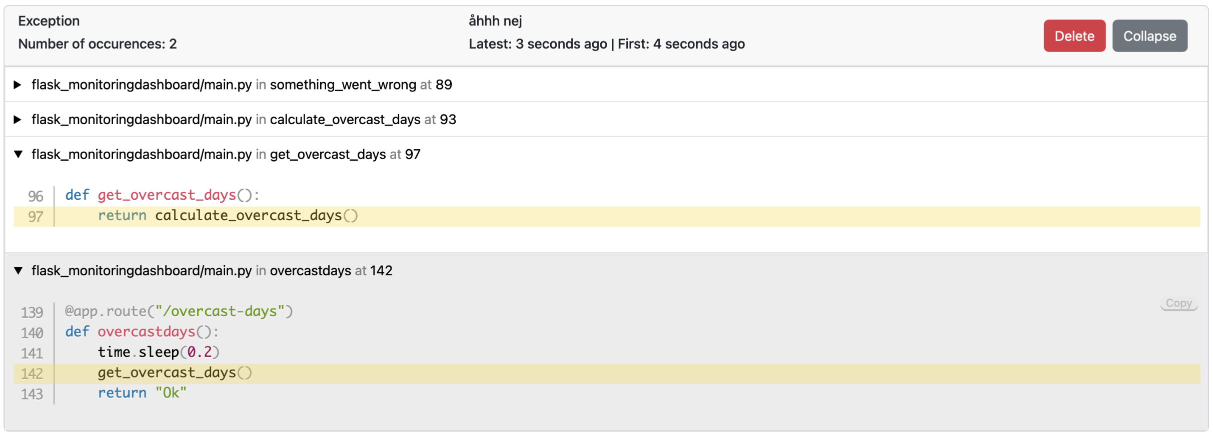
Displaying **1 - 2** exceptions: **2** in total

Endpoint details

Endpoint	overcastdays
Color	🟪
ID	12
Rule(s)	/overcast-days

Figure 6: Exceptions for the “overcastdays” endpoint

The endpoint specific exception page, provides an expandable view of the entire stack trace for an exception group (see Figure 6)[16]. Each stack line in this stack trace communicates the file name, function name, and line number of the occurred exception. Thereby, it mimics the application logs presented in Section 3.3. Additionally, each line can be expanded to show the corresponding code.



Exception — åhhh nej
Number of occurences: 2 — Latest: 3 seconds ago | First: 4 seconds ago

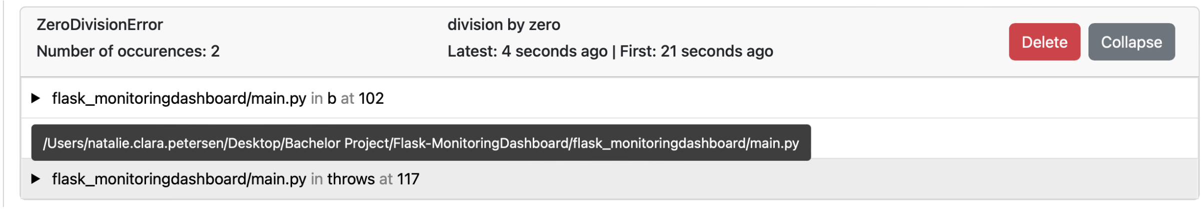
```
▸ flask_monitoringdashboard/main.py in something_went_wrong at 89
▸ flask_monitoringdashboard/main.py in calculate_overcast_days at 93
▾ flask_monitoringdashboard/main.py in get_overcast_days at 97

def get_overcast_days():
    return calculate_overcast_days()

▾ flask_monitoringdashboard/main.py in overcastdays at 142

@app.route("/overcast-days")
def overcastdays():
    time.sleep(0.2)
    get_overcast_days()
    return "Ok"
```

Figure 7: A single stack trace group with the corresponding source code

We show the function code with line numbers and a highlight of the exception invocation point; The line where the exception occurred or a function call was made to the next frame in the stack (See Figure 7).
To improve stack trace readability, we chose to display file paths relative to the application directory for all files inside the application. The full file path is still accessible by hovering

[16]In the case of Figure 6, there are only two exception groups, but we can display up to 5 at a time via pagination.

over the relative path and appears right above the relative path (see Figure 8). For files outside of the application (e.g. imported library) we still always show the entire file path.



ZeroDivisionError
Number of occurences: 2

division by zero
Latest: 4 seconds ago | First: 21 seconds ago

▸ flask_monitoringdashboard/main.py in b at 102

/Users/natalie.clara.petersen/Desktop/Bachelor Project/Flask-MonitoringDashboard/flask_monitoringdashboard/main.py

▸ flask_monitoringdashboard/main.py in throws at 117

Figure 8: Hovering over a file path (relative to the application) reveals the full absolute path.

An overarching philosophy behind the UI design was to provide the user with an overview presenting the most relevant information and to allow for elaboration on request. This can be seen in many places, such as the FMD's overview (see Figure 4). Here we have a counter with the amount of exceptions which users can learn more about when clicking on the number. The concept of progressive disclosure also presents itself when we click to fold out the stack trace code (see Figure 7), or when we hover to reveal the full file path (see Figure 8).

5.1.2 Functional Design

When implementing the frontend, we take the following observations into account: (1) Stack traces can be long; (2) The same function can appear multiple times in the same stack trace.

By default, no function code is displayed; users must expand each stack trace line to view the corresponding code. Therefore, regarding observation (1), we fetch code on demand, i.e. when the user chooses to expand the stack line. As for observation (2), caching seemed like the right solution as we can refer to a function via its id. When the code is requested, we put it in a hash table which is held in a local variable[17], i.e. we cache it for fast future access.

5.2 Modelling Exceptions in the Database

All exception-related data in our module is structured around requests, as exceptions only occur in relation to a request.

[17]This means that the cache is not persisted

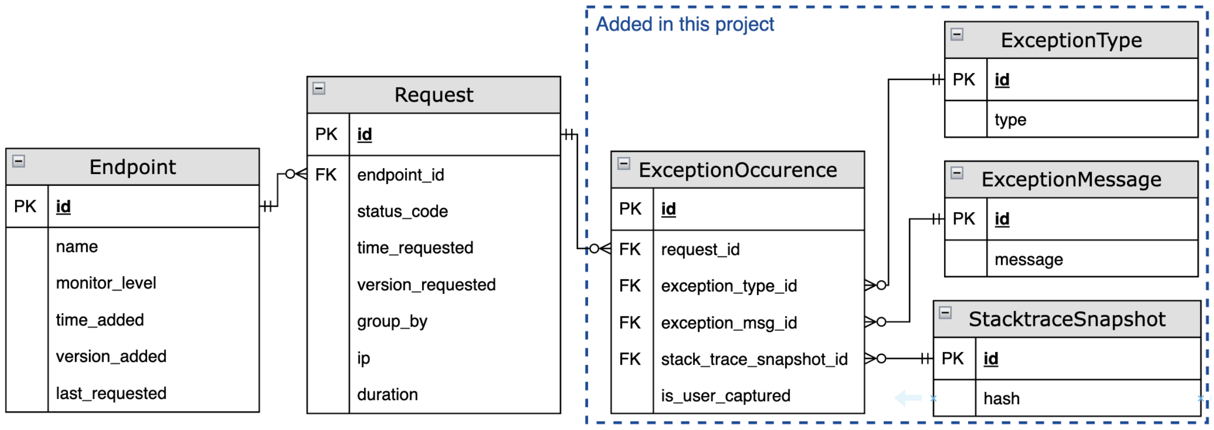


Figure 9: ExceptionOccurrence extending the Request table

Prior to this project, the database model contained the `Request` table representing HTTP requests. This table includes an endpoint_id referencing an entry in the `Endpoint` table, which holds information about the HTTP endpoints in the flask application. We extended this request model by introducing the table `ExceptionOccurrence` (see Figure 9). This table stores additional details when an exception occurs during a request, associating each exception with a specific exception type, message, and stack trace.

Important information about exceptions includes when they occurred and what endpoints were requested. The database already included `endpoint_id` and `time_requested` for each `Request`. Since all exceptions are associated with some `Request`, we do not need to include this information directly in `ExceptionOccurrence`, (see Figure 9).

`ExceptionOccurrence` entries refer to both an `ExceptionType` and an `ExceptionMessage`. These correspond to the type and message of Python exception objects. Additionally, `ExceptionOccurrence` also refers to a `StackTraceSnapshot` entry. This is used to associate an `ExceptionOccurrence`-entry with all the information about the stack trace of the exception. The `StackTraceSnapshot` ensures that a given stack trace is only modelled in detail *once* even if multiple exceptions have identical stack traces.

5.2.1 Storing Stack Trace Details

To extend the database model to support exception monitoring, we added nine new tables (See Figure 10); five besides the ones already introduced in Figure 9 were created to store information about the stack trace.

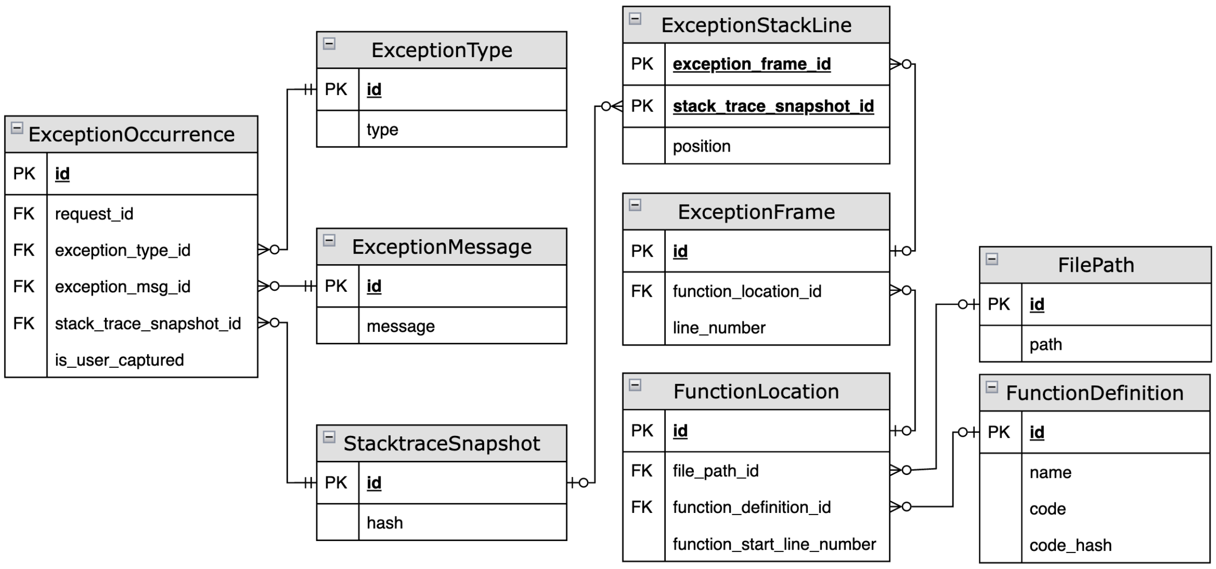


Figure 10: New exception database model

We store distinct stack traces as hashes in the table called `StacktraceSnapshot`. For each `FrameType` object in the `TracebackType`, we store an `ExceptionFrame`. The `ExceptionFrame` stores the exact line number relevant to that frame (see tb_lineno in Figure 3) and links to a `FunctionLocation`, which refers to more detailed information about the function, including the file path, starting line number, and reference to the actual function code. This means that an `ExceptionFrame` represents a specific stack frame in a stack trace for a specific version of a function.

Because stack frames must retain their original order, we also need to track their position within the stack trace. The same `ExceptionFrame` can appear in multiple stack traces at different positions. To handle this, we introduced the `ExceptionStackline` table, whose sole purpose is to associate each `ExceptionFrame` with a specific position in a given `StacktraceSnapshot`.

To store the related function code of a stack trace, we created a table called `FunctionDefinition`. A `FunctionDefinition` stores the related code as a string, along with its name, an ID, and a hash of the code. Storing the entire function code as a single string, as opposed to a string for each line of code, avoids the need for complex queries to reconstruct it, making it easier and faster for the frontend to retrieve and display the function.

The hash of the function code enables a more database-friendly way to check if an identical version of the function already exists in the database.

5.3 Implementation of Exception Grouping

5.3.1 The Motivation for Grouping

When storing exceptions, we save each occurrence separately, but presenting them this way to the user would most likely result in an overload of repeated information, causing important insights to drown and making recurring issues hard to recognize. Instead, we want to give the user a manageable overview. To achieve this, we want to group similar exceptions together.

5.3.2 How We Implemented Grouping

The key component of our grouping functionality is the `StackTraceSnapshot` table. All exceptions are linked to a `StackTraceSnapshot` according to a hash created from their stack trace. We group the exceptions by comparing these hashes.

To create the hash, we start by hashing the formatted stack trace[18]. This initial hash provides information on both the type and message of the exception, but also includes the order of the stack lines of the stack trace. Each stack trace line contains a reference to a location in the source code consisting of the file path, function name, and line number. However, the code on the referenced location might change over time, and since we are interested in the actual code that was executed to raise the given exception, we want the hash to depend on the specific source code at the time of the exception. Therefore, we let the hash combine the formatted stack trace with the code of the functions to which it refers.

```python
def get_function_definition_from_frame(frame: FrameType) -> FunctionDefinition:
    f_def = FunctionDefinition()
    f_def.code = inspect.getsource(frame.f_code)
    f_def.code_hash = text_hash(f_def.code)
    ...
    return f_def

def hash_stack_trace(exc: BaseException, tb: TracebackType) -> str:
    stack_trace_string = "".join(traceback.format_exception(type(exc), exc, tb))
    stack_trace_hash = text_hash(stack_trace_string)
    return _hash_traceback_type_object(stack_trace_hash, tb)


def _hash_traceback_type_object(h: str, tb: Union[TracebackType, None]) -> str:
    if tb is None:
        return h

    f_def = get_function_definition_from_frame(tb.tb_frame)
    new_hash = text_hash(h + f_def.code_hash)

    return _hash_traceback_type_object(new_hash, tb.tb_next)
```

Figure 11: Functions for obtaining the hash of a stack trace

To retrieve the function code, we access the frame of each `TracebackType` object. As we step through the `TracebackType` objects, we create a `FunctionDefinition` for each function, which also creates a hash of the function code. The hash of the function code is

[18] An example of how a formatted stack trace could look is shown in Appendix A

recursively concatenated with the previous hash, initially being the hash of the formatted stack trace. This is done until the None object is reached, indicating the end of the stack trace. The final hash will then be the result of both the initial formatted stack trace and the code of all the referenced functions. A code snippet from our implementation can be seen in Figure 11.

5.4 Capturing and Logging Exceptions

5.4.1 Collecting Exceptions

```
class ExceptionCollector:
    def __init__(self) -> None:
        self.user_captured_exceptions: list[BaseException] = []
        self.uncaught_exception: Union[BaseException, None] = None

    def add_user_captured_exc(self, e: BaseException):
        e_copy = _get_copy_of_exception(e)
        self.user_captured_exceptions.append(e_copy)

    def set_uncaught_exc(self, e: BaseException):
        e_copy = _get_copy_of_exception(e)
        self.uncaught_exception = e_copy
```

Figure 12: Code snippet from ExceptionCollector implementation

The class `ExceptionCollector` collects all exceptions that occur during a given request (see Figure 12). It holds copies of both uncaught and caught exceptions during the execution and persists them to the database once the request completes.

The ExceptionCollector is attached to Flask's request-scoped ”g” object, which is an object provided by Flask to store data during the lifecycle of a single request[19]. This is useful as it allows us to store multiple exceptions throughout a request.

Storing copies of the exceptions are important, since they might be re-raised later in the same request. We want to preserve them exactly as they were before being re-raised, without including any subsequent stack frames that might be added after the re-raise.

5.4.2 Logging Uncaught Exceptions

In FMD all the methods that handle the requests for a given endpoint are wrapped in functions that collect request-related monitoring data. Each monitoring level defines their own wrapper function.

Uncaught exceptions during a request are temporarily caught in the try-except block

[19]Flask's scoped global ”g” object. https://flask.palletsprojects.com/en/stable/appcontext/#storing-data (accessed: 10.05.2025)

and captured using the `set_uncaught_exc` method of the `ExceptionCollector` (see Figure 13). The try-except block ensures that all relevant FMD request data can be logged and saved in the database even when an exception occurs.

The data is written to the database using a separate thread and after logging, the exception is re-raised to avoid interfering with the application's logs.

```
@wraps(endpoint_method)
def wrapper(*args, **kwargs):
    g.e_collector = ExceptionCollector()

    try:
        result = endpoint_method(*args, **kwargs)
        ...
        exception = None
    except BaseException as e:
        g.e_collector.set_uncaught_exc(e)
        result = None
        ...
        exception = e

    ... # seperate thread logs result and collected exceptions to database

    if exception is not None:
        raise exception

    return result
```

Figure 13: A simplification of how endpoint methods are wrapped in FMD.

5.4.3 Logging User-Captured Exceptions

The FMD, by default, only logs uncaught exceptions [20], but by calling the `capture()` method of the dashboard object (see Figure 14), users are also able to log caught exception to the FMD database.

```
import flask_monitoringdashboard as dashboard
try:
    ...
except Exception as e:
    dashboard.capture(e)
```

Figure 14: Our capture exception API [21]

[20]unless monitoring level 0 is selected, then nothing is logged for the endpoint
[21]This api mimics the api used by other exception logging tools (see Appendix D).

```
    def capture(e: Exception):
        from flask import g
        if "e_collector" not in g:
            g.e_collector = ExceptionCollector()
        g.e_collector.add_user_captured_exc(e)
```

Figure 15: Our capture exception implementation

The `capture()` function uses the `ExceptionCollector` to "capture" the exception via the `add_user_captured_exc` method (see Figure 15).

5.5 Data Pruning

FMD users can trigger data pruning by specifying the number of weeks of data they wish to retain; data older than this threshold will be deleted. To maintain consistency and prevent excessive data accumulation, we extended this pruning mechanism to include exception data.

Additionally, we introduced a manual deletion option that allows users to delete specific exception groups individually, through a dedicated button in the interface. This is relevant if a user identifies the reason for a certain exception and corrects the code to fix the problem so that the exception no longer occurs. The ability to delete specific groups of exceptions provides users with more control over their stored exception data while maintaining the simplicity of the overall pruning system.

6 Discussion

In Section 5 we have shown that you can implement an exception monitoring module without requiring extra user configuration and without overly complex modifications. Thus, we answer our research question:

> "How can the Flask Monitoring Dashboard be effectively extended to incorporate exception monitoring without user configuration, providing a unified self-hosted solution for Flask monitoring?"

What we describe is an implementation in the Flask Monitoring Dashboard, but most of the concepts can be used for monitoring services in general, not only for Flask monitoring.

In the following, we will elaborate on some of the thoughts that the implementation gave rise to.

6.1 When to Log Exceptions

Monitoring level 1, 2, and 3 track status codes, and display a small red number indicating requests with status codes above 400. By integrating exception logging into these monitoring levels, we provide context to the red numbers, helping users better understand the source of the issues and how they might relate to performance problems. With this integration we preserve that no data is stored in the database at monitoring level 0, but users can now not monitor performance (use monitoring level 1, 2 or 3) without also tracking exceptions.

Instead of adding exception logging to these existing levels, an alternative could have been to enable or disable exception monitoring of an endpoint with a toggle, allowing users to choose whether to include it independently of the monitoring levels. However, we believe that users will always want to know what happened if an exception occurs on a monitored endpoint.

6.2 Security and Privacy Considerations

6.2.1 Logging Source Code

In general, depending on the user, logging the source code could be an issue for exception tracking frameworks, since it exposes their intellectual property. This is a consideration that other frameworks like Sentry address by being able to opt out of source code logging[22]. This is a key consideration for them, as the source code logged to their server contains someone else's Intellectual Property.

Before this thesis, the FMD only logged source code as a result of profiling, which is done on monitoring level 3, but now we monitor exceptions on every monitoring level except 0. Thereby source code logging is no longer exclusive to monitoring level 3. For this reason, it is relevant to reflect upon whether the user should be able to opt out of source code logging, as they can in Sentry. This option is very important for products like Sentry which saves users code externally, however, when using the FMD, the users control their data themselves[23]. The user might still want the ability to disable source code logging as they just might not care about viewing the source code, and prefer a quick overview about exceptions on the specific endpoints. Therefore, this is an improvement that could be considered for the future and would provide users with more granular control over their data.

6.2.2 Logging Request Bodies

Logging request bodies might lead to faster troubleshooting. However, it could lead to privacy issues, as request bodies might contain personally identifiable information, passwords, cookies, or other sensitive information, and we would have no way of knowing what information is sensitive. Of course, there is a solution to this, but it would require us to build a small framework around request data filtering, where users could specify which sensitive data to remove or obfuscate. We did not include it, but it might be an idea for the future.

6.2.3 Logging Variables

Similarly to request bodies, variables could also contain sensitive information, and therefore logging of variables would also require some way of filtering to avoid privacy issues.

[22]Ability to opt out of Source Context in Sentry. https://docs.sentry.io/platforms/python/data-management/data-collected/#source-context (accessed: 07.04.2025)

[23]Thus not being forced to expose their IP to anyone but themselves

The stack trace might pass through some functions where we would like to log the variables and some where we would not. This could be fixed by providing some configuration for the `capture` method, but unfortunately this was beyond our scope.

6.3 Considerations on the UI Design

When choosing what to display on the frontend, we want to achieve a balance between providing comprehensive information and clarity, ensuring that relevant details are presented without overwhelming the users with information.

6.3.1 Displaying Function Bodies

We believe that the ability to view the stack trace of an exception is essential for fast debugging, as it allows developers to actively view what events occurred leading up to an error. To fully understand the events, we believe that simply knowing the location in the code (file name, function name, and line number) is not enough. Storing the actual code referenced by the stack trace is essential, as it preserves a snapshot of the codebase at the time of the error. This ensures that even if the code changes later, the exact state of the application when the error occurred can still be accessed, providing valuable context for debugging and resolution.

A simple solution to this issue would be to include a link to the latest git commit at the time of the request, e.g. retrieved via git blame[24]. This would ensure that the user would be able to access the specific version of the source code that was executed when the exception occurred. But this approach might be misleading, as the changes made in the specific commit were not necessarily the ones that introduced the problem. The source of the issue might already have been included several commits earlier, but just not revealed itself until then.

Instead, we chose to display the relevant code directly in the dashboard, and by including the timestamp for when the latest exception occurred, the user will still be able to identify which commits occurred before.

The idea of displaying code, referenced by the stack trace, directly in the dashboard is seen in one of the existing features of Sentry (see Appendix C), but unlike Sentry, we chose to log the entire function body rather than just a snippet of it.

Limiting the display to a few lines before and after the error ensures relevant context while avoiding excessive verbosity. However, this approach comes with the downside that some important context might be missed if the error is influenced by code outside the displayed range. We want the user to be able to get as much of the relevant information as possible from the FMD, avoiding the necessity of going into the logs or source code. Showing the entire function provides more comprehensive debugging information, even though it may introduce unnecessary noise. A future optimization could be to minimize the default view to display only a code snippet, with the ability to expand on demand.

In our implementation, the function bodies are presented with syntax highlighting. Al-

[24]Git blame documentation. https://git-scm.com/docs/git-blame (accessed: 10.05.2025)

though studies have not found any evidence that syntax highlighting eases code comprehension (Beelders and Plessis 2016 [BD10]), we still decided to use it, as it reflects the design in modern text editors.

6.3.2 Managing Large Stack Traces

A significant challenge in stack trace visualization is determining how much information to display. Often, some lines of the stack trace will be less relevant than others. Exceptions occurring at the end of recursive function calls, for example, often produce long stack traces with duplicated lines, leading to unnecessary clutter. A potential solution to this issue is to group these recursive lines and indicate the repetition with a counter. However, this approach becomes complex when dealing with mutual recursion or cases where the next recursive call does not always occur on the same line.

Another problem is deep call-stacks, which can obscure the most relevant parts of the stack trace, making debugging more difficult. This calls for some way of limiting the stack trace length.

Standard logging tools often truncate stack traces, while raw stack trace objects capture full details. To strike a balance, one approach is to display the initial and final sections of a stack trace while omitting the middle portion, ensuring visibility of both the error location and the originating call. The first few lines provide insight into where the exception originated, while the last few lines indicate the exact point at which the error was raised. This approach would ensure the presentation of the stack trace to be interpretable, but consequently also less informative. To avoid missing information, the ability to expand the view to access more details might be a good idea.

We believe that implementing some handling of large stack traces, providing overview, while preserving meaningful context, would enhance the usability of the Flask Monitoring Dashboard significantly, but given the complexity of designing a good algorithm for this, we unfortunately did not find time to include a solution in our implementation. It remains an area for potential future refinement.

6.4 Considerations on Database Storage

Prior to this project, the FMD already contained a data model supporting the storage of code ordered in a stack. This model associates `StackLine`s with a `CodeLine` object and a `Request` object while maintaining an explicit position to order the lines (see Figure 16). Although this structure works for other features in FMD, it is not ideal for comparing full stack traces, which is a key requirement for our exception grouping functionality.

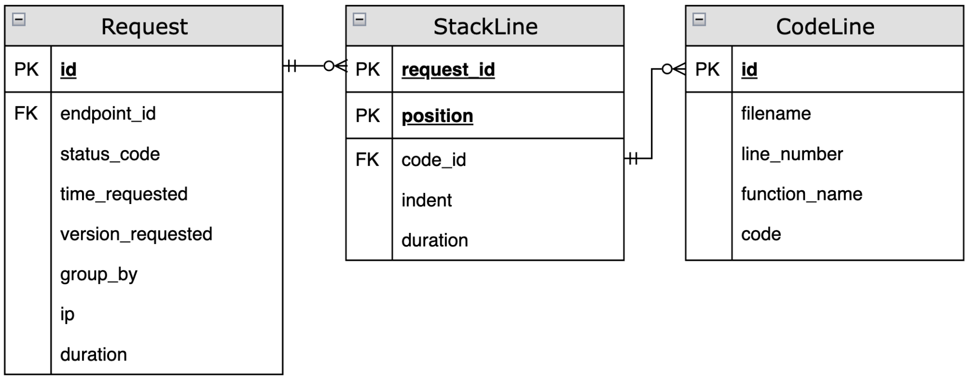


Figure 16: The CodeLine and StackLine table in relation to a Request

6.4.1 Considerations on Storing Stack Traces

Our model for exception stack traces is similar to the one for the other features of the FMD, as it also needs to track code, its order in the stack and the request it belongs to. The `ExceptionStackLine` table is comparable to the `StackLine` table as they both store the position of a frame/line in the stack and reference the corresponding code. However, a key distinction is how they connect to the rest of the data model.

In the original design, each `StackLine` is linked directly to a `Request`, but if we used this structure for the `ExceptionStackLine`s, we would not be able to distinguish the stack traces of different exceptions thrown in the same request. If we linked them to an exception instead of a request, this problem would be solved, but it would still be difficult to compare entire sets of `ExceptionStackLine`s i.e. stack traces. Comparing two stack traces would require iterating through all stack lines associated with each request, reconstructing and comparing them one line at a time.

To address this, we introduced the `StacktraceSnapshot` table. Instead of linking each `ExceptionStackLine` directly to an exception, it references a `StacktraceSnapshot` entity which is also referenced by one or more `ExceptionOccurrence`s. This allows the same `StacktraceSnapshot` entity to be reused for multiple exceptions, possibly thrown at multiple requests, if the stack trace they create is identical. It also enables us to treat a full stack trace as a single identifiable entity; instead of comparing lists of stack lines, we can now compare snapshots directly.

6.4.2 Considerations on Storing Code

In theory, saving code line by line would normalize the database, but the `CodeLine` table is not normalized, and additionally, using it would result in multiple practical complications; each time a new function appears in a stack trace, we would need to parse it line by line, check for existing matches in the `CodeLine` table, and carefully avoid storing duplicates. Furthermore, displaying a function in the frontend would require reconstructing it from many individual lines.

The `CodeLine` model stores the source code together with its location, consisting of the file path together with the line number in the same file. If code is moved between files, new entries of the code would be stored. Similarly, a single shift of the line numbers in the file, unrelated to the functionality of the line of code, would result in a cascading effect causing every succeeding line of code to be re-stored, even if the succeeding code itself remains unchanged.

Our implementation of the exception module requires many comparisons between function bodies. In addition, lines of code are never displayed individually. Therefore, we decided to store a string of the entire function instead of splitting it into individual code lines. Additionally, we decided to store a hash to use when comparing functions[25]. Whether the combination of hashing first and then comparing is more efficient than the direct comparison without hashing remains to be determined.

6.5 Challenge of Exception Grouping

To meaningfully group exceptions, we must evaluate and decide which factors indicate that two exceptions should be considered the same. Our philosophy is to rather provide slightly more information than too little, ensuring users get enough context directly in the UI to avoid unnecessary trips to the logs. This means that we must strike a balance between grouping related exceptions while still preserving key differences that might be relevant for debugging.

In the following we will present different key factors that influenced our solution for grouping.

6.5.1 Grouping by Exception Type and Message

While the type and message provide context, the same exception type with the same exception message can be thrown at different places in the code and/or be triggered by different execution paths. Since these are separate issues that might require individual fixes, we should avoid grouping them together as a single problem.

Another consideration is that messages may contain variable states that do not necessarily indicate a different issue. For example, if the user specifies an exception message with context-specific details such as in Figure 17.

```python
raise ValueError(f"Age must be non-negative integer: {age}")
```

Figure 17: Exception with context specific details

Since the message may differ, we treat them as distinct issues to simplify frontend display and ensure more granular error tracking, as the user may include context-specific details for a reason.

6.5.2 Grouping by Stack Trace

Looking at the stack trace helps determine whether two exceptions follow the same execution path; however, when grouping, relying solely on stack traces formatted as in the

[25]It should be noted that we have not measured the speed difference between comparing the functions directly and comparing their hashes, but an argument could be made that hash comparison is generally faster

application logs can be problematic.

The stack lines in the stack trace only consider the file path, line number, function name, and the line of code which raised the exception. Meaning, if an exception occurred at a specific line in the source code, future exceptions happening at the same line would have identical stack traces, even if related code has changed.

```python
def somefunc(i):
    num = 2
    return i/(i*num)
```

(a)

```python
def somefunc(i):
    num = 0
    return i/(i*num)
```

(b)

Figure 18: Example of undetected code changes

For instance, if the code block in Figure 18a is changed to 18b, but the division causes a ZeroDivisionError both before and after the change, the stack trace will not be different (see Figure 19), i.e. a `ZeroDivisionError` exception would occur if ”`i`” was zero in 18a, but would in 18b occur every time as ”`num`” is zero.

```
File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
    flask_monitoringdashboard/main.py", line 145, in somefunc
return i / (i * num)
       ~~~~~~~~~~~~~
ZeroDivisionError: division by zero
```

Figure 19: ZeroDivisionError stack trace in the application log

Conversely, if line numbers shift due to minor edits (e.g., comments or formatting changes not related to the problematic function), two identical exceptions might appear different when they are not, since they would have different stack traces. This is a limitation to be aware of, but it is not a problem we have solved in our implementation.

6.5.3 Grouping by Code Version

When grouping exceptions, code changes must be considered. The same exception type and formatted stack trace can be derived from different underlying code, if the code has been modified without affecting the file name, function name or line number where the exception occurred (see Figure 18/19). To ensure accurate debugging, we store the actual code referenced in each stack trace, preserving a snapshot of the codebase at the time the error occurred.

Our UI displays the entire source code for each function in the stack trace, allowing developers to inspect the precise logic that led to the exception, even if the code has since changed. Exceptions are grouped only when both the stack trace and the code in the relevant functions are identical. Grouping exceptions with identical formatted stack traces but differing underlying function code risks obscuring the true cause, as seemingly similar errors may stem from entirely different issues. Choosing to group this way would also complicate the UI, as each stack frame could have a different code version within the same exception group. To make grouping reliable, we store and compare hashes of entire function bodies for each stack frame, as we have already described in Section 5.3.

Another approach for determining the code version could be to compare git commit hashes instead of hashing the executed functions. But grouping by commits would most likely separate exceptions unnecessarily. Different groups of exceptions could be formed, even when the executed code has not changed, simply because a commit unrelated to the function was made. By hashing the executed functions directly, we avoid this issue and ensure that exception groups reflect meaningful differences in code behaviour.

6.5.4 Interdependence Between Frontend Design and Grouping

The classification of the exception groups is directly affected by the properties presented in the UI. Therefore, it is necessary to consider what information will be displayed on the front; the more properties we display in the UI, the more properties should be considered when grouping. As we choose to display the message of an exception and not just the type, we might need two distinct presentations of exceptions with identical types if the messages were not identical. The same applies when we extend the display of the stack trace to include the function code; exceptions with identical stack traces might require different presentations if the functions have been updated.

We group exceptions with the purpose of simplifying the presentation of exceptions and thereby gives the user a better overview. If the properties of ten elements in a list are identical, there is no point in presenting it ten times; the user will get the same information if it is presented once, including a note saying this is the case for ten incidences.
As the exceptions are not completely identical but only similar, handling them as a group means that there are some unique properties which will not be visible. In our solution, this limitation includes information about the time of occurrence.

Exceptions are grouped according to properties that locate the source of the problem and identify the kind of exception. We display the timestamp of the first and latest occurrence of exceptions in the group, but not each exception timestamp. This gives a sense of the time span over which the exception has appeared but hides trends in frequency. As a consequence of this, users cannot tell whether the exception is becoming more or less frequent. An improvement to our solution could be to include some visualisation of the frequency, e.g. a graph showing occurrences over time.

7 Concluding Remarks

The FMD is a useful tool for performance monitoring, but prior to this project lacked any kind of exception tracking, forcing the users to either rely on logs or external tools to get stack trace information. The goal of this thesis was to extend FMD with exception monitoring, which users should receive “for free” and with minimal configuration.

To achieve this, we implemented a database structure which supports the logging of exceptions and relating these to specific requests. Using this structure, FMD now supports the logging of caught and uncaught exceptions mimicking existing exception logging frameworks. The dashboard presents the information about the exceptions grouped by hashes created depending on their stack trace and the function code related to their stack trace. Implementing the exception monitoring functionality has not introduced any configuration or migration requirements for the users.

7.1 Further Work

While this project lays the foundation for exception monitoring capabilities within the FMD, there are still many possibilities for future features that would improve the debugging experience.

In section 6.2, we discussed the ability to log application state, but concluded that it was not feasible in the scope of this project. This feature could be considered in future work.

We spent considerable time on the stack trace visualization, but as mentioned in section 6.3.2, when long stack traces are presented, they can be cumbersome to examine. This suggests that, future work could include research on improving the frontend design for better readability and understanding of stack traces.

Finally, as mentioned in Section 6.5.4, another promising direction for future work, is to visualize the trends of the exceptions over time. In our implementation, we only show the first and most recent occurrence of a group of exceptions. This provides a time range but no indication of how often the exceptions occur within a certain period of that range. Adding support for frequency tracking (e.g. a graph visualization) could help users identify tendencies in occurrences over time.

8 Source Code

The source code for this project, including our changes, can be found at GitHub on the master branch

https://github.com/flask-dashboard/Flask-MonitoringDashboard

and on PyPI (Flask-MonitoringDashboard v4.0.1)

https://pypi.org/project/Flask-MonitoringDashboard/4.0.1

References

[Vog+17] Patrick Vogel et al. “A low-effort analytics platform for visualizing evolving Flask-based Python web services”. In: *2017 IEEE Working Conference on Software Visualization (VISSOFT)*. IEEE. 2017, pp. 109–113.

[Che19] An Ran Chen. “An Empirical Study on Leveraging Logs for Debugging Production Failures”. In: *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. 2019, pp. 126–128. DOI: 10.1109/ICSE-Companion.2019.00055.

[Pet19] Bogdan Petre. “Improving the Performance of a Performance Monitor. The case of the Flask Monitoring Dashboard.” PhD thesis. 2019.

[BD10] Tanya R Beelders and Jean-Pierre L Du Plessis. “Syntax highlighting as an influencing factor when reading and comprehending source code”. In: *Journal of eye movement research.* 9.1 (2016-02-10). ISSN: 1995-8692.

A Python Exception Example

```
Traceback (most recent call last):
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/env/lib/python3.13/
      site-packages/flask/app.py", line 1511, in wsgi_app
    response = self.full_dispatch_request()
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/env/lib/python3.13/
      site-packages/flask/app.py", line 919, in full_dispatch_request
    rv = self.handle_user_exception(e)
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/env/lib/python3.13/
      site-packages/flask/app.py", line 917, in full_dispatch_request
    rv = self.dispatch_request()
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/env/lib/python3.13/
      site-packages/flask/app.py", line 902, in dispatch_request
    return self.ensure_sync(self.view_functions[rule.endpoint])(**view_args) #
        type: ignore[no-any-return]
           ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~^
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
      flask_monitoringdashboard/core/measurement.py", line 151, in wrapper
    raise e_logger.uncaught_exception_info
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
      flask_monitoringdashboard/core/measurement.py", line 121, in evaluate_
    result = route_handler(*args, **kwargs)
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
      flask_monitoringdashboard/main.py", line 116, in throws
    d()()
    ~~~~^
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
      flask_monitoringdashboard/main.py", line 106, in c
    return b()
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
      flask_monitoringdashboard/main.py", line 102, in b
    return a()
  File "/Users/albert/pythonProjs/Flask-MonitoringDashboard/
      flask_monitoringdashboard/main.py", line 98, in a
    raise Exception("ohhh no!")
Exception: ohhh no!
```

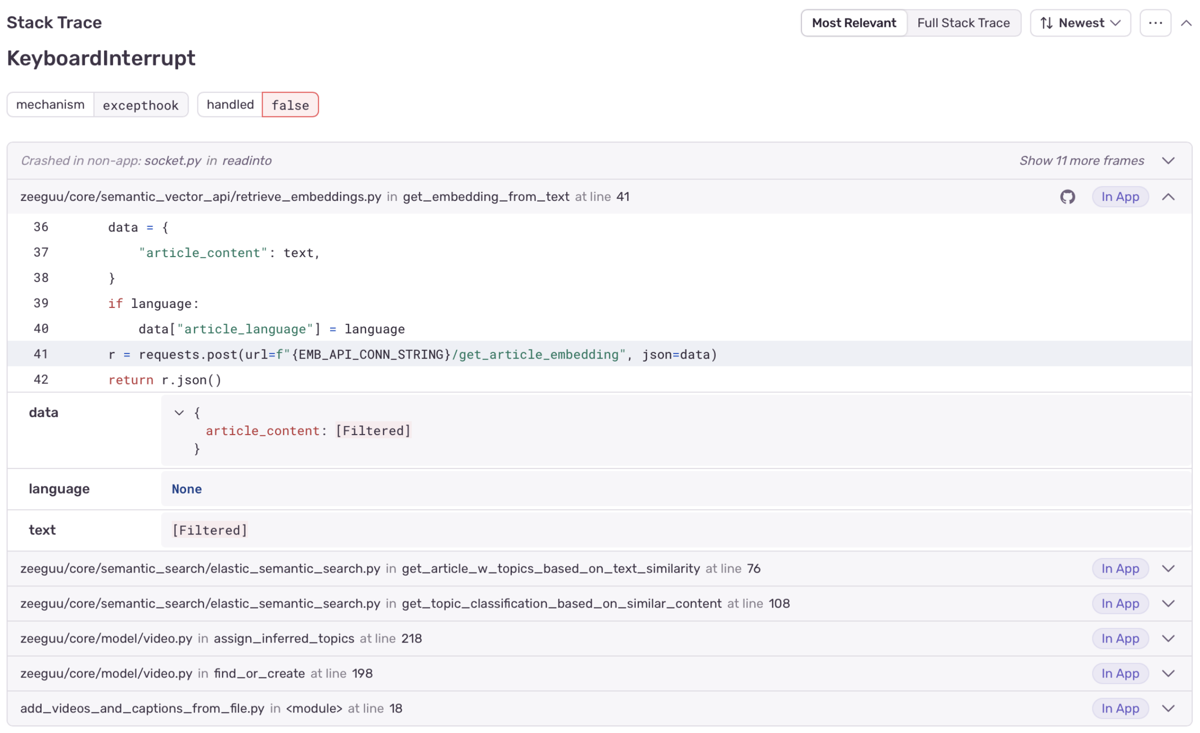
Figure 20: Simple stacktrace logged by python

B Error Tracker Grouping implementation

```python
def get_context_detail(
        request,
        masking,
        context_builder,
        additional_context):

    ty, val, tb = sys.exc_info()
    frames = traceback.format_exception(ty, val, tb)
    traceback_str = format_exception(tb, masking=masking)
    frame_str = ''.join(frames)
    rhash = sha256(str.encode(frame_str, "UTF-8")).hexdigest()
    request_data = (
        context_builder.get_context(
            request,
            masking=masking,
            additional_context=additional_context)
    )
    return ty, frames, frame_str, traceback_str, rhash, request_data
```

Figure 21: Error tracker grouping, code reformatted for readability

C How Sentry Displays Stack Traces



KeyboardInterrupt

mechanism excepthook | handled false

Crashed in non-app: socket.py in readinto

zeeguu/core/semantic_vector_api/retrieve_embeddings.py in get_embedding_from_text at line 41

```
36        data = {
37            "article_content": text,
38        }
39        if language:
40            data["article_language"] = language
41        r = requests.post(url=f"{EMB_API_CONN_STRING}/get_article_embedding", json=data)
42        return r.json()
```

data	{ article_content: [Filtered] }
language	None
text	[Filtered]

zeeguu/core/semantic_search/elastic_semantic_search.py in get_article_w_topics_based_on_text_similarity at line 76
zeeguu/core/semantic_search/elastic_semantic_search.py in get_topic_classification_based_on_similar_content at line 108
zeeguu/core/model/video.py in assign_inferred_topics at line 218
zeeguu/core/model/video.py in find_or_create at line 198
add_videos_and_captions_from_file.py in <module> at line 18

Figure 22: A stack trace in sentry, the source code shown is from an open source project with the MIT-License

D Capturing Exceptions in Other Frameworks

```python
import logging
try:
    ...
except Exception as e:
    logging.exception(e)
```

(a) Pythons logging module

```python
import bugsnag
try:
    ...
except Exception as e:
    bugsnag.notify(e)
```

(b) Bugsnags logging module

```python
import sentry_sdk
try:
    ...
except Exception as e:
    sentry_sdk.capture_exception(e)
```

(c) Sentry's logging module

Figure 23: User exception logging examples [26]

[26] Sentry capture exception: https://docs.sentry.io/product/sentry-basics/integrate-backend/capturing-errors/#capture-exception (accessed: 24.03.2025),
Pythons logging framework: https://docs.python.org/3/library/logging.html#logging.Logger.exception (accessed: 24.03.2025),
Bugsnags logging framework: https://docs.bugsnag.com/platforms/python/other/reporting-handled-errors/ (accessed: 24.03.2025)