pyconau

Developing Culture To Write Reliable and Performant Services At Scale

Harshit Prasad / harshitprasad.com

Introduction

- Software Engineer at Blinkit
- From 🇮🇳
- Tech Speaker
- Open Source Contributor
- Past 2 x GSoC

Why we write software?

To solve problems by making life easier for the people.

Good software implies:

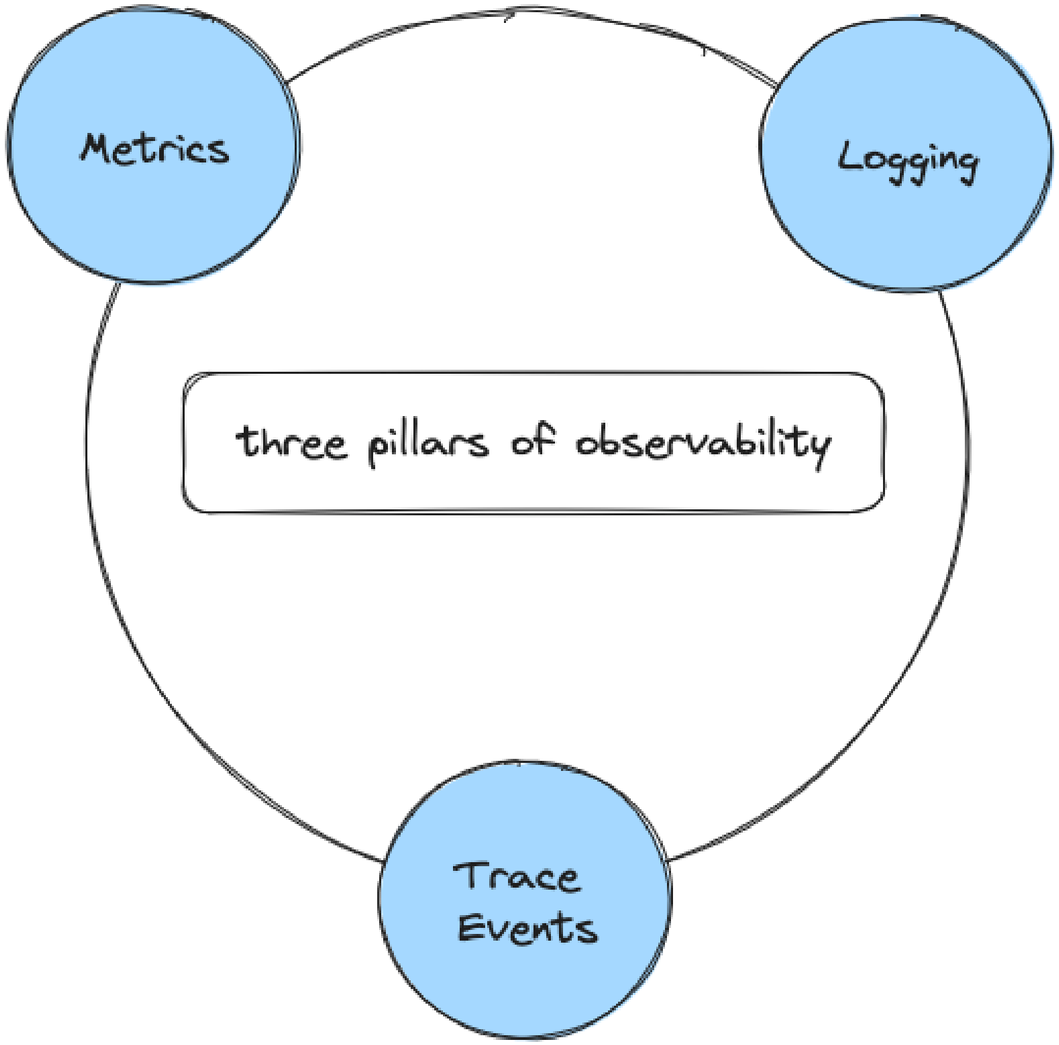
- Good amount of user traffic and engagement.
- Business Profits

Bad software implies:

- Downtimes
- Business Loss

Monitoring != Observability

Observability



Logging


debug
info
warn
error


Module
Timestamp
Process ID
Log Level
Message

```python
import logging

from helpers.log_formatter import log_formatter


class PyConLoggerFactory:

    def __init__(self):
        self.logger = logging.getLogger()
        self.stream_handler = logging.StreamHandler()

        self.__log_formatter = log_formatter

        self.__configure_logger()
        self.__configure_stream_handler()
        self.__configure_log_formatter()

    def __configure_logger(self):
        self.logger.setLevel(logging.INFO)

    def __configure_stream_handler(self):
        self.stream_handler.setLevel(logging.INFO)

    def __configure_log_formatter(self):
        self.stream_handler.setFormatter(self.__log_formatter)
        self.logger.addHandler(self.stream_handler)

    def get_logger(self):
        return self.logger


pylogger = PyConLoggerFactory()

```

```
[main]:2023/08/06:22:06:29 - 72231 | INFO: Testing pycon logger with INFO level
[main]:2023/08/06:22:06:29 - 72231 | WARNING: Testing pycon logger with WARN level
[main]:2023/08/06:22:06:29 - 72231 | ERROR: Testing pycon logger with ERROR level
```

```
{
  "timestamp": "2023/08/06:22:12:09",
  "level": "INFO",
  "message": "Testing pycon logger with INFO level",
  "module": "main",
  "line_no": 12,
  "process_id": 72711
}
{
  "timestamp": "2023/08/06:22:12:09",
  "level": "WARNING",
  "message": "Testing pycon logger with WARN level",
  "module": "main",
  "line_no": 13,
  "process_id": 72711
}
{
  "timestamp": "2023/08/06:22:12:09",
  "level": "ERROR",
  "message": "Testing pycon logger with ERROR level",
  "module": "main",
  "line_no": 14,
  "process_id": 72711
}
```

```python
import multiprocessing

from helpers.request_context import request_context
from pycon_logger.factory import pylogger

logger = pylogger.get_logger()


def process_requests(request_name):
    logger.info(f"Processing {request_name} with DEBUG level")
    logger.error(f"Processing {request_name} with INFO level")


def worker(request_name):
    request_context.generate_trace_id()
    process_requests(request_name)


def simulate_requests():
    with multiprocessing.Pool(processes=2) as worker_pool:
        worker_pool.starmap(worker, [("Request 1",), ("Request 2",)])


if __name__ == "__main__":
    simulate_requests()

```


```
{
  "timestamp": "2023/08/18:21:26:25",
  "level": "INFO",
  "message": "Processing Request 1 with INFO level",
  "module": "main",
  "line_no": 35,
  "process_id": 12981,
  "trace_id": "480c201f-8203-491e-9f6c-261f1fc1f549"
}
{
  "timestamp": "2023/08/18:21:26:25",
  "level": "ERROR",
  "message": "Processing Request 1 with ERROR level",
  "module": "main",
  "line_no": 36,
  "process_id": 12981,
  "trace_id": "480c201f-8203-491e-9f6c-261f1fc1f549"
}
{
  "timestamp": "2023/08/18:21:26:25",
  "level": "INFO",
  "message": "Processing Request 2 with INFO level",
  "module": "main",
  "line_no": 35,
  "process_id": 12981,
  "trace_id": "d789b48c-57db-429e-a90a-8ecd155e4820"
}
{
  "timestamp": "2023/08/18:21:26:25",
  "level": "ERROR",
  "message": "Processing Request 2 with ERROR level",
  "module": "main",
  "line_no": 36,
  "process_id": 12981,
  "trace_id": "d789b48c-57db-429e-a90a-8ecd155e4820"
}
```

Limitations of Logging

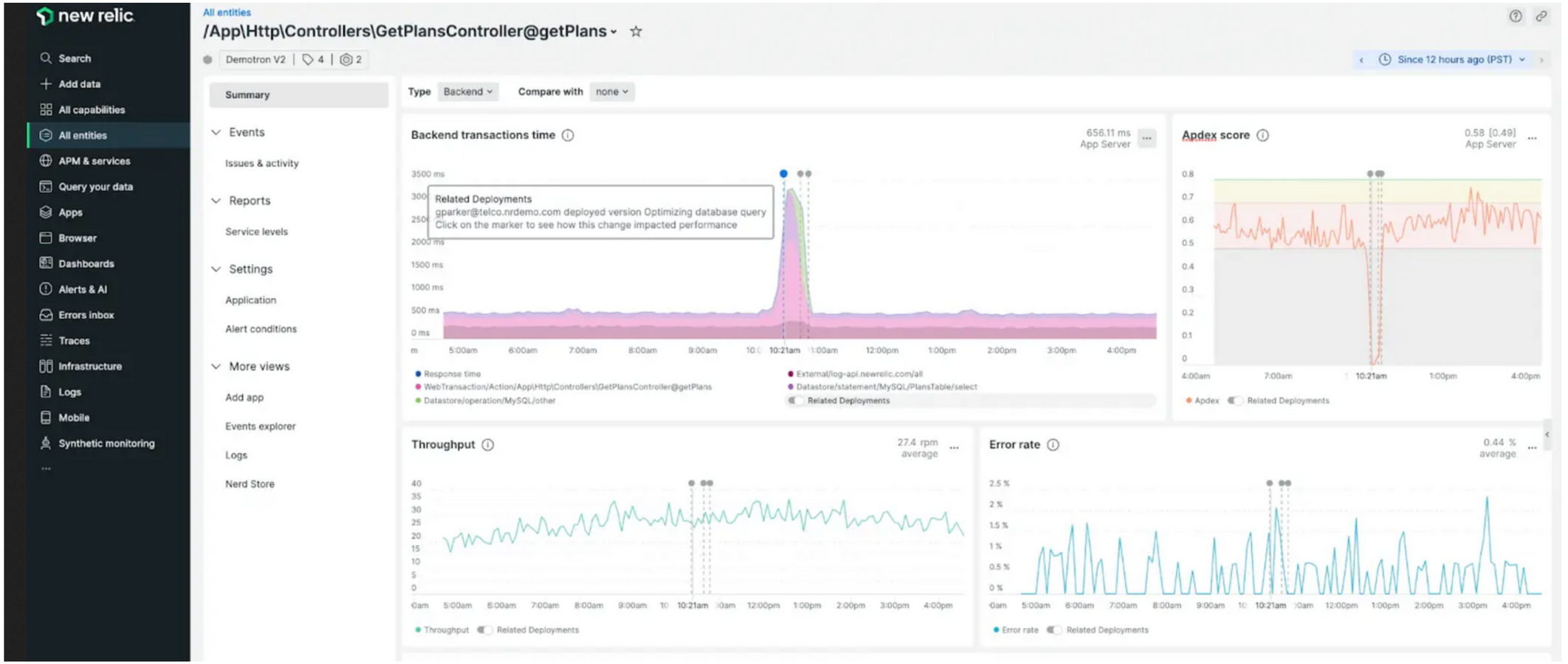
- Extensive logging can generate large volumes of data leading to storage challenges.
- Careless logging practices can lead to information leakage.
- Log Noise.
- Does not provide quantitative measurement of system behaviour.

Metrics

Four golden signals important for your software

- Latency
- Traffic Throughput
- Error Rate
- Saturation

Example


new relic
Search
Add data
All capabilities
All entities
APM & services
Query your data
Apps
Browser
Dashboards
Alerts & AI
Errors inbox
Traces
Infrastructure
Logs
Mobile
Synthetic monitoring
All entities
/App\Http\Controllers\GetPlansController@getPlans
Demotron V2 | 4 | 2
Since 12 hours ago (PST)
Summary
Events
Issues & activity
Reports
Service levels
Settings
Application
Alert conditions
More views
Add app
Events explorer
Logs
Nerd Store
Type Backend
Compare with none
Backend transactions time
656.11 ms
App Server
Related Deployments
gparker@telco.nrdemo.com deployed version Optimizing database query
Click on the marker to see how this change impacted performance
Response time
WebTransaction/Action/App\Http\Controllers\GetPlansController@getPlans
Datastore/operation/MySQL/other
External/log-api.newrelic.com/all
Datastore/statement/MySQL/PlansTable/select
Related Deployments
Apdex score
0.58 [0.49]
App Server
Apdex
Related Deployments
Throughput
27.4 rpm
average
Throughput
Related Deployments
Error rate
0.44 %
average
Error rate
Related Deployments

Example

```python
import newrelic.agent

from redis_client import redis_client


HALL_A_CONF_ROOM = 'hall_a'
HALL_B_CONF_ROOM = 'hall_b'
HALL_C_CONF_ROOM = 'hall_c'


class PyConConfHallManager:

    @newrelic.agent.function_trace('PyConConfHallManager.occupied')
    def occupied(self, hall_key):
        return redis_client.get(hall_key)

    @newrelic.agent.function_trace('PyConConfHallManager.book')
    def book(self, hall_key):
        redis_client.set(hall_key, True, expiration=60 * 30)  # 30 mins

```

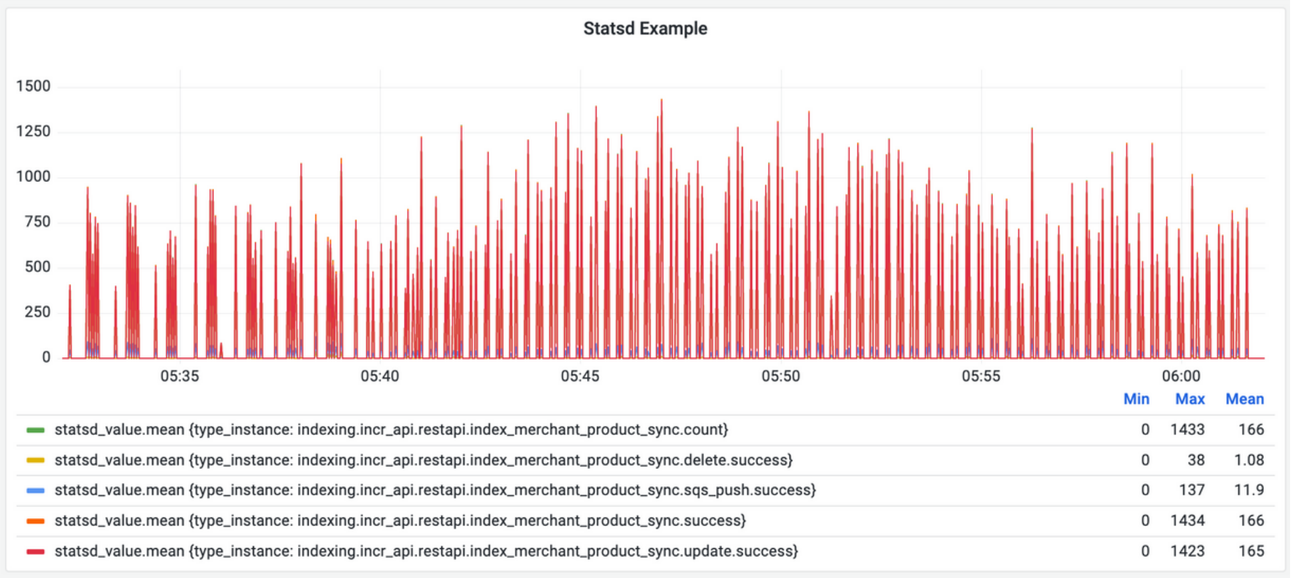
Breakdown table

Category	Segment	↓ % Time	Avg Calls/Txn	Avg Time
Database	Redis get	3.78	7.02	15.8 ms
Database	Redis set	2.84	4.13	11.9 ms

Example

```python
def success_fail_metric_count():
    try:
        # Start timing the request processing
        start_time = time.time()

        # Simulate processing a request
        time.sleep(1)
        elapsed_time = time.time() - start_time

        statsd_client.timing(REQUEST_PROCESSING_TIME_KEY, elapsed_time * 1000) # in ms
        statsd_client.incr(SUCCESSFUL_REQUEST_COUNT_KEY)

    except Exception as e:
        statsd_client.incr(FAILED_REQUEST_COUNT_KEY)


def latency_calc():
    try:
        with statsd_client.timer(REQUEST_LATENCY_KEY):
            time.sleep(2.3)
            print("I just woke up!")
    except Exception as e:
        statsd_client.incr(FAILED_REQUEST_COUNT_KEY)
```

SELECT mean("value") FROM "statsd_value" WHERE ("type" = 'count' AND "type_instance" =~ /^indexing.incr_api.restapi*/) AND $timeFilter GROUP BY time($__interval), "type_instance" fill(0)



Limitations of Metrics

- Lack of detail and provides limited context only.
- Metric overload.
- Over optimization.

Events

Events are fundamental component of the observability but, they slightly provide a different purpose compared to logs and metrics.

- Context rich information
- Metadata and structured data like timestamps, event types, identifiers, and additional attributes.
- Helps in reconstructing the history of the system behavior.

Example

```sql
create type action_type as enum ('ORDER_PLACED', 'ORDER_CANCELLED');

create type ticket_type as enum ('STUDENT', 'PROFESSIONAL', 'HOBBYIST');

create table pycon_booking_events (
    id serial primary key not null,
    user_id int not null,
    event_type varchar(255) not null,
    action action_type not null,
    ticket_type ticket_type not null,
    timestamp timestamp with time zone not null default current_timestamp,
)

```

Example

```python
import psycopg2

from enum import Enum
from datetime import datetime


class Action(Enum):
    ORDER_PLACED = 'ORDER_PLACED'
    ORDER_CANCELLED = 'ORDER_CANCELLED'


class TicketType(Enum):
    STUDENT = 'STUDENT'
    PROFESSIONAL = 'PROFESSIONAL'
    HOBBYIST = 'HOBBYIST'


TABLE_NAME = "pycon_booking_events"

db_connection = psycopg2.connect(
    dbname='postgres',
    user='admin',
    password='admin',
    host='localhost',
    port=6432
)
```

```python
class PyconTicketBookingEvent:

    def __init__(self, request_context, timestamp=None):
        self._event_type = request_context.get("event_type")
        self._user_id = request_context.get("user_id")
        self._action = None
        self._ticket_type = None
        self._timestamp = timestamp or datetime.now()

        self._table_name = TABLE_NAME
        self._conn = db_connection

```

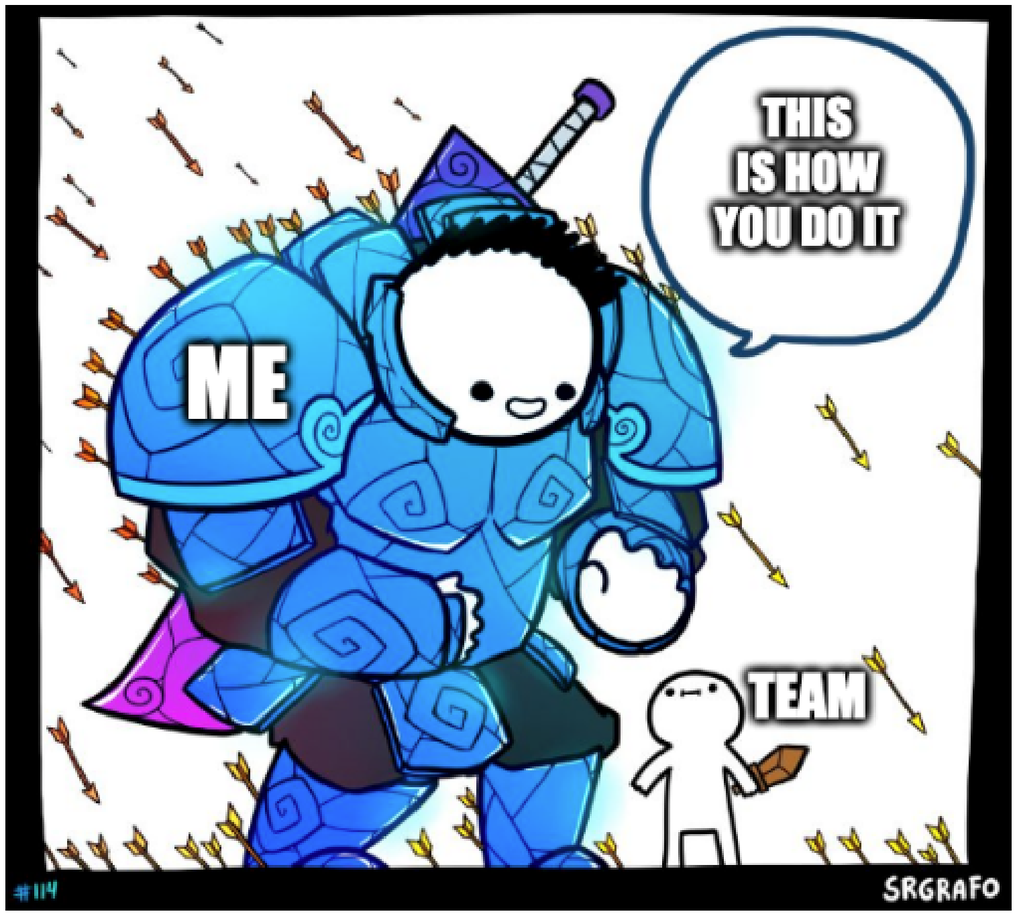
Example

```python
    def set_event_attributes(self, action, ticket_type):
        self._action = action
        self._ticket_type = ticket_type

    def emit(self):
        event_json = self.__get_event_dict()

        event_type = event_json.get("event_type")
        action = event_json.get("action")
        timestamp = event_json.get("timestamp")
        ticket_type = event_json.get("ticket_type")
        user_id = event_json.get("user_id")

        with self._conn.cursor() as cursor:
            cursor.execute(
                f"insert into {self._table_name} (user_id, event_type, action, ticket_type, timestamp) VALUES (%s, %s, %s, %s, %s)",
                (user_id, event_type, action, ticket_type, timestamp,)
            )
            self._conn.commit()

    def __get_event_dict(self):
        return {
            'event_type': self._event_type,
            'action': self._action,
            'timestamp': self._timestamp,
            'ticket_type': self._ticket_type,
            'user_id': self._user_id,
        }
```

```python
    event = PyconTicketBookingEvent(request_context=request_context)
    event.set_event_attributes(action, ticket_type)
    event.emit()
```

Observability Driven Culture

"Culture is the way you think, act and interact"



Educate the team

Using Right Tools and Standardize Events format

Post Incident Reviews

Lead by Example and Celebrate Success!

Thank You!

harshit98

harshit-prasad

HarshitPrasad8

harshitprasad.com