# Adaptive Tracing
# Problematic Area Localization

Masoumeh Nourollahi

Polytechnique Montréal

DORSAL Laboratory

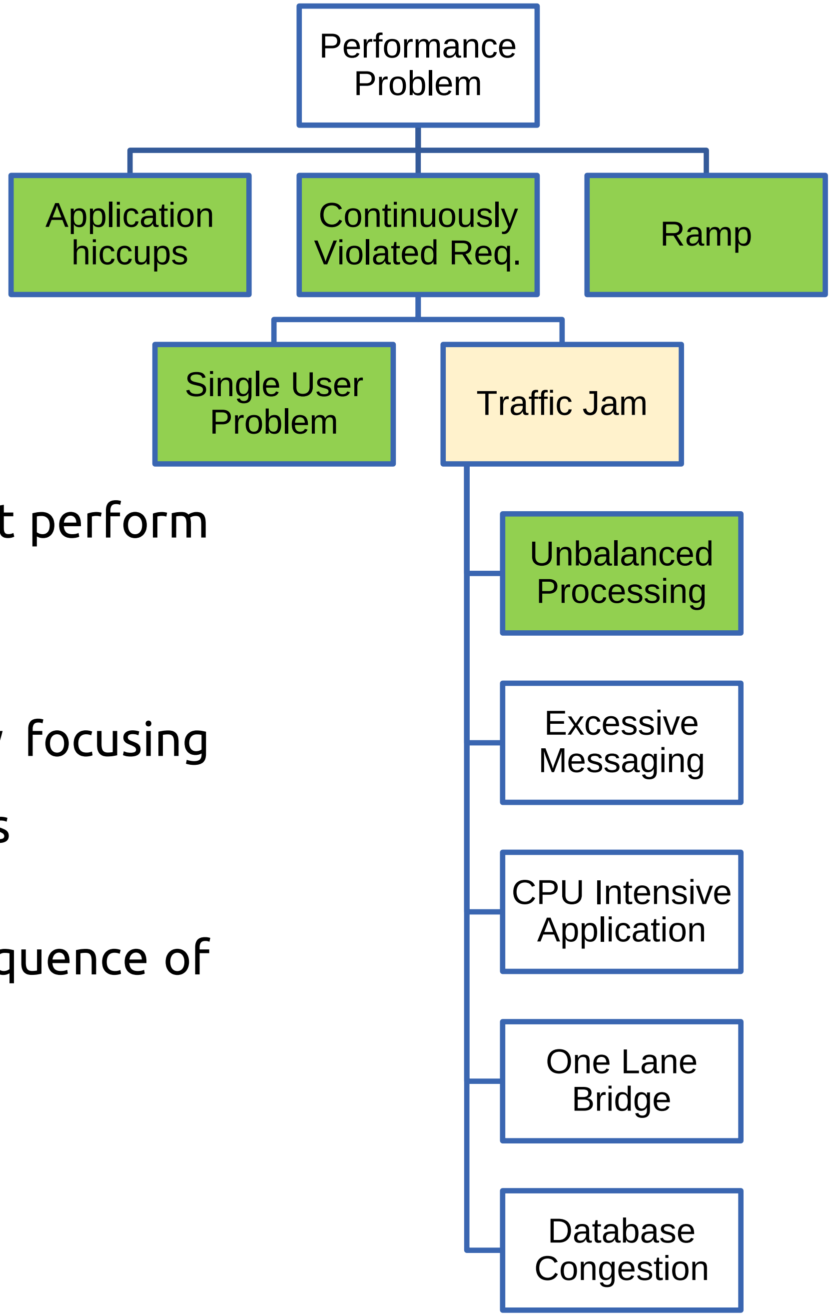
# Agenda

- Research Questions
- Performance problems
- Methodology
- Test setup
- Experiments
    - General performance problem
    - Hiccups
    - Ramp
    - Traffic jam
- Conclusions

# Research Questions

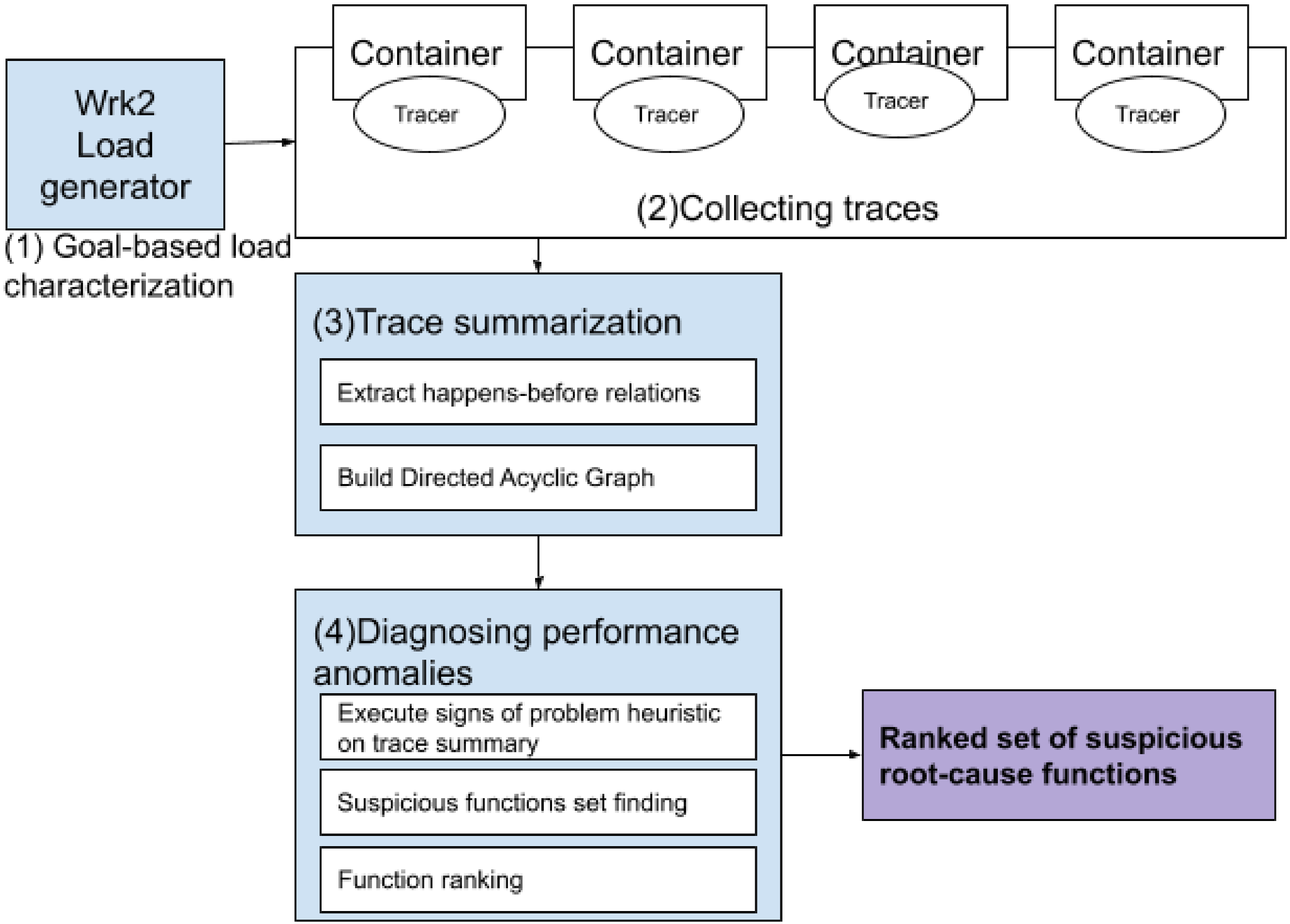
1. Goal based tracing impact on controlling tracing overhead and efficiency
   - By limiting the number of required tracepoints at the start of tracing
2. Determining which tracepoints to activate for goal based tracing
3. Localizing problematic areas in the system, with consecutive tracing adaptation goal in mind
   - By detecting signs of performance anti-patterns

# Performance Problems Taxonomy




- Determine if the application generally does not perform well
- Identify patterns of performance changes, by focusing on a specific category of performance problems
- Improve root-cause analysis by locating the sequence of function executions containing the pattern

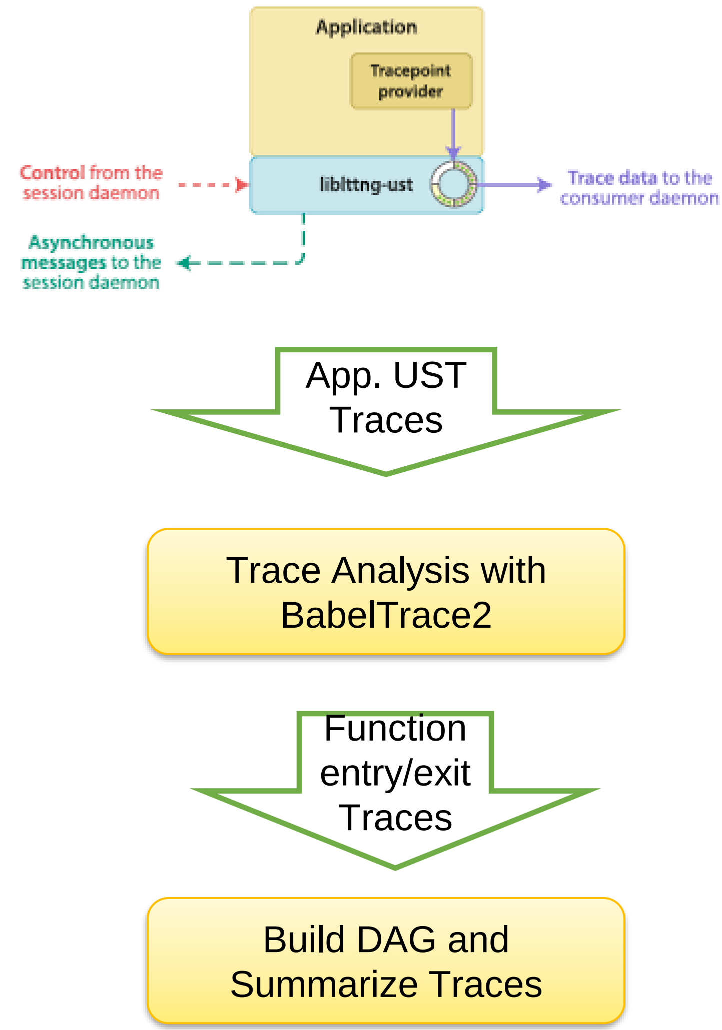
# Methodology



# Goal-based Tracing

## Looking for signs of performance problems

- Looking into signs of performance anti-patterns as the starting point of application tracing
- Provides initial set of candidates for detailed tracing
- Previous work on kernel tracing helps determine main places we should focus on, like candidate services
- Next level is function entry/exit tracing
- The call graph can be constructed from happens-before relations to track root-cause in the chain of events



# Tracepoint event and probe samples

```
TRACEPOINT_EVENT(
    masoum_tp,
    ComposePostServiceClient_FUNC_send_UploadUserMentions,
    TP_ARGS(
        const char* , event_type,
        const char* , file_name,
        const char* , func_name,
        int , loc),
    TP_FIELDS(
        ctf_string(event_type_field, event_type)
        ctf_string(file_name_field, file_name)
        ctf_string(func_name_field, func_name)
        ctf_integer(int, loc_field, loc)
   )
)
```

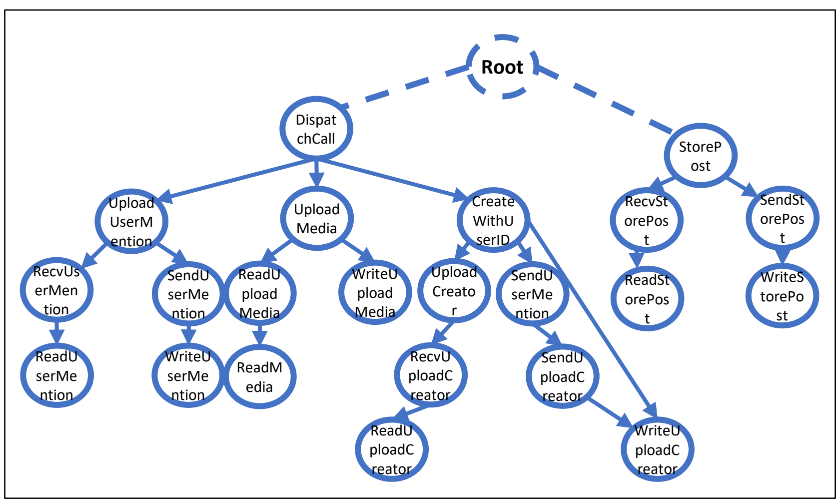
```
void ComposePostServiceClient::send_UploadUserMentions(const int64_t req_id, const std::vector<UserMention> & user_mentions, const std::map<std::string,
std::string> & carrier)
{
  tracepoint(masoum_tp, ComposePostServiceClient_FUNC_send_UploadUserMentions, "en", __FILE__, __FUNCTION__, __LINE__);
  int32_t cseqid = 0;
  oprot_->writeMessageBegin("UploadUserMentions", ::apache::thrift::protocol::T_CALL, cseqid);

  ComposePostService_UploadUserMentions_pargs args;
  args.req_id = &req_id;
  args.user_mentions = &user_mentions;
  args.carrier = &carrier;
  args.write(oprot_);

  oprot_->writeMessageEnd();
  oprot_->getTransport()->writeEnd();
  oprot_->getTransport()->flush();
  tracepoint(masoum_tp, ComposePostServiceClient_FUNC_send_UploadUserMentions, "ex", __FILE__, __FUNCTION__, __LINE__);
}
```

# Load characterization and Trace Summarization

$event_i$= (funcId, eventType, timeStamp, ThreadId)

$func_i$= (funcId, callerId, entryTs, exitTs, duration, threadId)

$edge_i$= (edgeId, sourceId, targetId, entryTs, exitTs, duration)




**Algorithm 1** Build function happens-before relationship directed acyclic graph

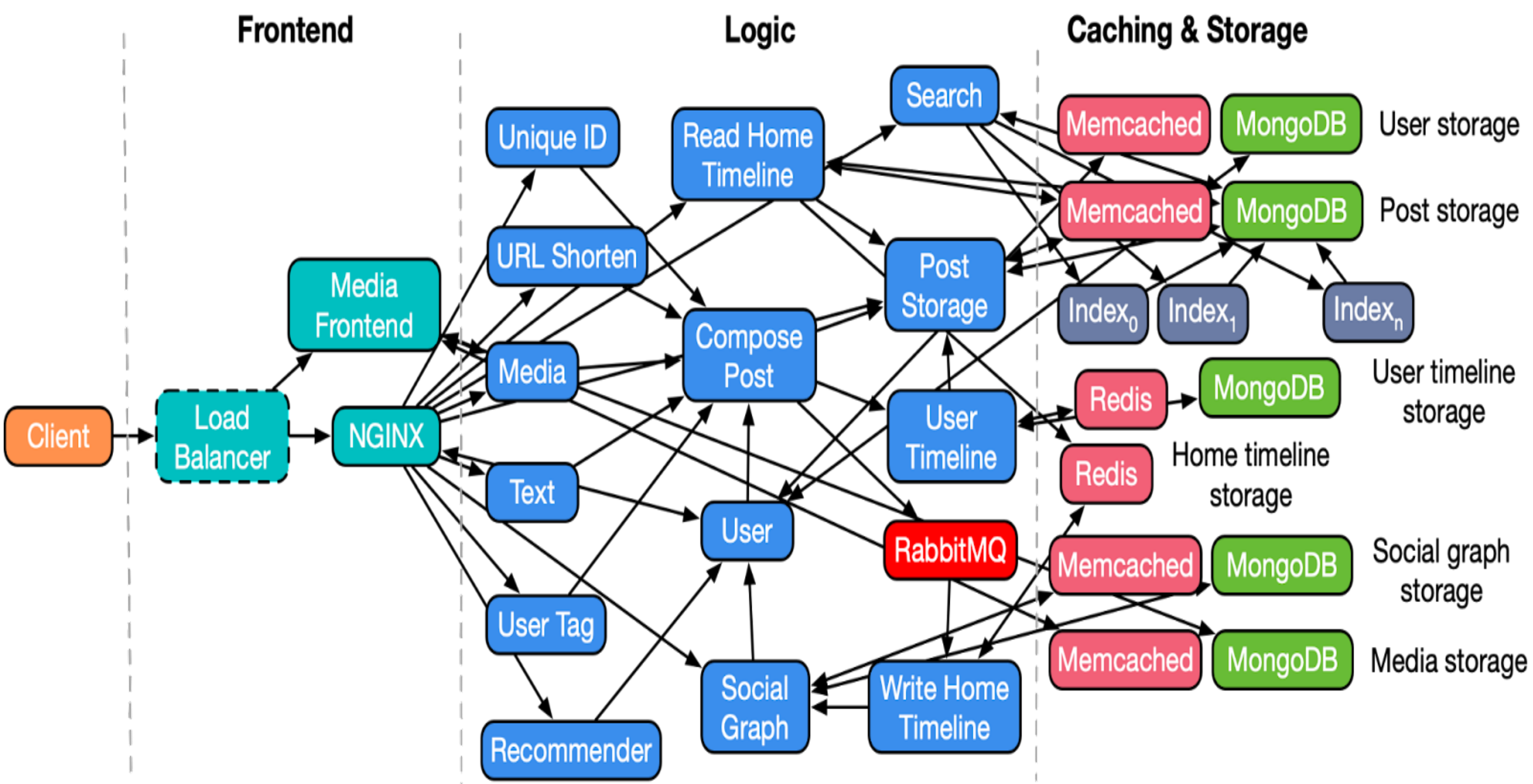
```
T − func_i ← function − entry/function − exittraces
list − edge = []
list − vertex = []
for Trace_i do
    Filter events based on threadId
    for threadId_i do
        stack_i = []
        for event_k do
            if event_ktype ← func − entry then
                Push stack_i ← event_k
                caller − func ← event_k
            end if
            if event_ktype == func − exit then
                Pop
                callee − func ← event_k
            end if
            entry_ts ← caller − entry_ts
            exit_ts ← callee − exit_ts
            list − edge_i ← caller − func, callee − func, entry_ts, exit_ts
            list − vertex_i ← caller, callee
            response − time_i ← (callee − exit_ts(i)) − (caller − entry_ts(i))
        end for
    end for
    Build directed acyclic graph(list-vertex, list-edge)
    for edge_k do
        vertex_k ← caller − func
        vertex_k+1 ← callee − func
        Graph ← edge_k, response − time_i, vertex_k, vertex_k+1
        Calculate weight edge_k
    end for
end for
```

# Diagnosing performance anomalies

| Anti-pattern | Test service | Fault Injection method | Number of traces |
|---|---|---|---|
| **Generic performance Problem** | Compose Post | Delayed code by adding a long time function | 100,000 |
| **Hiccup** | User Timeline | Delay in 20% of cases on “WriteUserTimeline” function | 100,000 |
| **Traffic Jam** | Compose Post | Changed code to not release lock mutex | 100,000 |
| **Ramp** | Compose Post | No changes | 100,000 |

# Test setup

# DeathStarBench (SocialNetwork)- test setup



# DeathstarBench- Social Network Service

- Supported actions in DeathStarBench
  - Create text post (optional media: image, video, shortened URL, user tag)
  - Read post
  - Read entire user timeline
  - Receive recommendations on which users to follow
  - Search database for user or post
  - Register/Login using user credentials
  - Follow/Unfollow user

| Service | Total LoCs | Communication Protocol | Unique Microservices | Per-language LoC breakdown |
|---|---|---|---|---|
| Social Network | 68061 | RPC | 36 | 34% C, 23% C++, 18% Java, 7% node.js, 6% Python, 5% Scala, 3% PHP, 2% Javascript, 2% Go |

Code Composition

# Experiments

# Experiment 1: Generic performance problem

**Algorithm 2** Generic Performance Problem detection

Trace Characterization

Run heavy load on the intended service
$Traces \leftarrow Collectfunction - entry/function - exittraces$
$caller - callee - relation \leftarrow Algorithm1(Traces)$

Execute signs of problem heuristic on trace summary

**for** $caller - callee - relation_i$ **do**
  Calculate $mean - duration - edge_i \leftarrow mean(response - time_i)$
  **for** $edge_i$ **do**
    $Threshold - mid_i \leftarrow mean - duration - edge_i$
    $Threshold - factors \leftarrow 10^{-4}, 10^{-3}, 10^{-2}, 10^{-1}, 1, 10^1, 10^2, 10^3, 10^4$
    **for** $Threshold - factor_k$ **do**
      $Threshold_{ik} \leftarrow (Threshold - mid_i) * (Threshold - factor_k)$
    **end for**
  **end for**
**end for**
**for** $edge_i$ **do**
  **for** $Threshold_{ik}$ **do**
    **if** $response - time_i < Threshold_{ik}$ **then**
      $has - problem - flag_{ik} \leftarrow False$
    **else**
      $has - problem - flag_{ik} \leftarrow True$
    **end if**
  **end for**
  $Cumulative_i \leftarrow size(has - problem - flag_i == True$
**end for**

Suspicious functions set finding

$Cumulative - Ratio_i \leftarrow (Cumulative_i)/(size_i)$
$Cumulative - Ratio - set \leftarrow Cumulative - Ratio_i$

Ranking suspicious functions

$Performance - problem - ranking \leftarrow Sort(Cumulative - Ratio - set)$

# General performance problem outputs

- Set of suspicious function calls
- Automatically finding the performance requirements for each function pair (edge), that is required in other problems analysis

# Experiment Results

| **Test case** | **Load intensity** | **Services tested** | **Number of test functions** | **Functions ranked as problematic** |
|---|---|---|---|---|
| Generic performance problem- Base Load | 1 Thread<br>1 Connection<br>10 Seconds<br>1 Requests/Second | read-user-timeline<br>compose-post<br>read-home-timeline | 195 | No problems observed |
| Generic performance problem- Normal Load | 5 Thread<br>5 Connections<br>10 Seconds<br>5 Requests/Second | read-user-timeline<br>compose-post<br>read-home-timeline | 195 | Next slide |
| Generic performance problem- Stress Load | 500 Thread<br>500 Connections<br>10 Seconds<br>2000 Requests/Second | read-user-timeline<br>compose-post<br>read-home-timeline | 228 | Next slide |

# Experiment Results- Ranking suspicious functions

| # | (Service1, Service2) | Edge (Func1, Func2) | Probability |
|---|---|---|---|
| 1 | (composepost_processor, composepost_UploadUserMentions_args) | (process_UploadUserMentions, read) | 0.5 |
| 2 | (composepost_processor, composepost_UploadUrls_args) | (UploadUrl, read) | 0.5 |
| 3 | (composepost_processor, composepost_UploadUrls_result) | (process_UploadUrl, write) | 0.5 |
| 4 | (composepost_processor, composepost_UploadUserMentions_result) | (process_UploadUserMentions, write) | 0.5 |
| 5 | (composepost_client, composepost_processor) | (dispatchCall, process_UploadUrls) | 0.5 |
| 6 | (composepost_client, composepost_processor) | (dispatchCall, process_UploadUserMentions) | 0.5 |
| 7 | (composepost_UploadUrls_args, socialnetworktypes_Url) | (read, read) | 0.5 |
| 8 | (composepost_UploadUserMentions_args, socialnetworktypes_UserMention) | (read, read) | 0.5 |
| 9 | (composepost_processor, composepost_UploadUniqueId_args) | (process_UploadUniqueId, read) | 0.24 |
| 10 | (composepost_UploadCreator_args, socialnetworktypes_Creator) | (read, read) | 0.09 |

# Experiment 2: Hiccup performance problem

- Inputs
  - Response time series of each thread functions
  - Threshold T that shows max response time acceptable for that function
  - Maximum allowed proportion the cumulative hiccup time may cover, of the whole experiment duration
- Output
  - Set of functions that are hiccup candidates

**Algorithm 3** Hiccup Performance Problem detection

```
Run normal load on the intended service
Traces ← Collectfunction − entry/function − exittraces
caller − callee − relation ← Algorithm1(Traces)
Calculate performance requirement threshold
for caller − callee − relation_i do
    Calculate mean − duration − edge_i ← mean(response − time_i)
    for edge_i do
        Threshold − mid_i ← mean − duration − edge_i
        Threshold − factors ← 10^-4, 10^-3, 10^-2, 10^-1, 1, 10^1, 10^2, 10^3, 10^4
        for Threshold − factor_k do
            Threshold_ik ← (Threshold − mid_i) * (Threshold − factor_k)
        end for
    end for
end for
Calculate moving percentile size
for edge_i do
    for edge_ik do
        occurrence − diff_k = entry − timestamp(edge_ik+1) − entry −
timestamp(edge_ik)
        occurrence − diff_i ← occurrence − diff_k
        occurrence − mean_i ← mean(occurrence − diff_i)
        moving − percentile_i ← (occurrence − mean_i) * (number − of −
occurrences)
    end for
    for Threshold_k do
        for moving − percentile_i do .............
        end for
    end for
    Cumulative_i ← size(has − problem − flag_i == True
end for
Cumulative − Ratio_i ← (Cumulative_i)/(size_i)
Cumulative − Ratio − set ← Cumulative − Ratio_i
Performance − problem − ranking ← Sort(Cumulative − Ratio − set)
```

# Experiment Results

| Test case | Load intensity | Services tested | # of test functions | Functions ranked as problematic |
|---|---|---|---|---|
| Hiccup performance problem- Normal Load | 5 Thread<br>5 Connections<br>10 Seconds<br>5 Requests/Second | read-user-timeline<br>compose-post<br>read-home-timeline | 195 | Next slide |
| Hiccup performance problem- Heavy Load | 500 Thread<br>500 Connections<br>10 Seconds<br>2000 Requests/Second | read-user-timeline<br>compose-post<br>read-home-timeline | 228 | Next slide |

Normal and Stress load shows the same results for Hiccup problem identification!

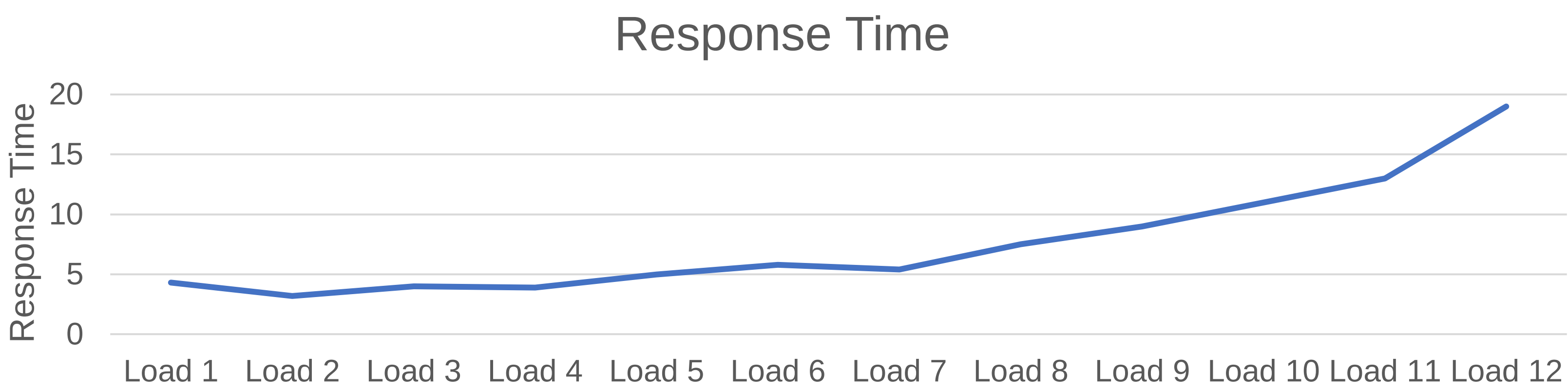
# Experiment Results- Ranking suspicious functions

| # | (Service1, Service2) | Edge (Func1, Func2) | Probability |
|---|---|---|---|
| 1 | (usertimeline_client, usertimeline_client) | (WriteUserTimeline, recv_WriteUserTimeline) | 0.27 |
| 2 | (poststorageservice_client, poststorageservice_client) | (StorePost, recv_StorePost) | 0.1265 |
| 3 | (usertimeline_client, usertimeline_presult) | (recv_WriteUserTimeline, read) | 0.01711 |
| 4 | (composepost_client, composepost_processor) | (dispatchCall, process_UploadUniqueId) | 0.0147 |
| 5 | (composepost_processor, composepost_UploadCreator_args) | (process_UploadCreator, read) | 0.0075 |
| 6 | (composepost_processor, composepost_UploadUniqueId_args) | (process_UploadUniqueId, read) | 0.0040 |
| 7 | (composepost_processor, composepost_UploadText_args) | (process_UploadText, read) | 0.0030 |
| 8 | (composepost_client, composepost_client) | (dispatchCall, dispatchCall) | 0.0024 |
| 9 | (composepost_client, composepost_processor) | (dispatchCall,process_UploadCreator) | 0.0024 |
| 10 | (composepost_client, composepost_processor) | (dispatchCall, process_UploadText) | 0.0009 |

# Experiment 3: Ramp performance problem- Time Window Strategy

There is an occurence of ramp, if a significant difference between response time of the beginning and end of the load test is observed

1. For each path/func a pairwise comparison of the response time series of neighbouring experiments is conducted
2. For this comparison response time sets for each experiment is gathered and bootstrapped to be normally distributed. Then a t-test is applied to compare
3. If comparison of starting and ending sets results in significant growth of response time, then ramp anti-pattern is observed



# Experiment Results

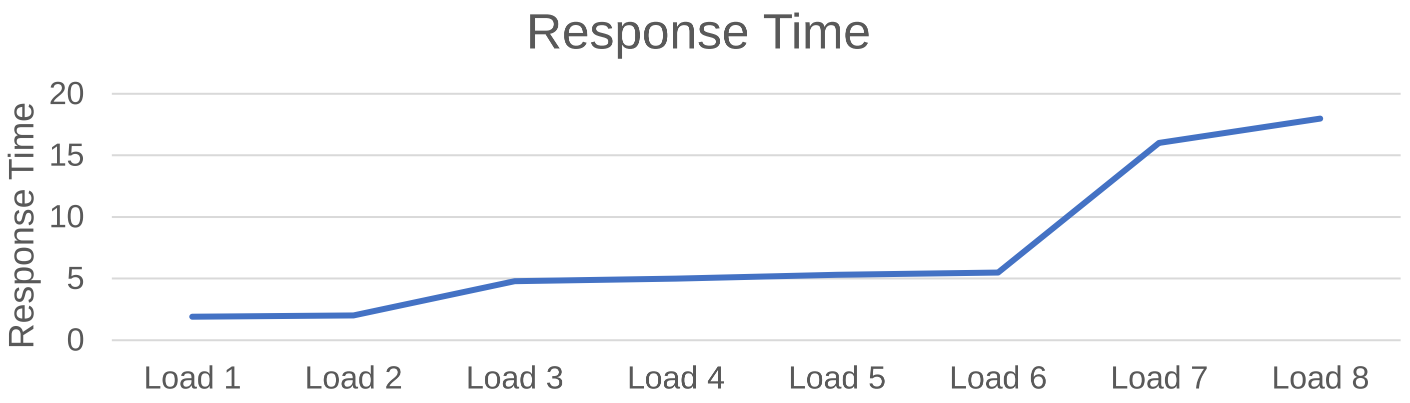
| Test case | Load intensity | Services tested | # of test functions | (Service, Function) ranked as problematic |
|---|---|---|---|---|
| Ramp performance problem- Constant Heavy Load | ● 5 intervals of putting heavy load and then gathering traces in base load<br>Base load configuration:<br>1 Thread, 1 Connections, 10 Seconds, 5 Requests/Second<br><br>● Heavy load configuration:<br>200 Thread, 200 Connections, 120 Seconds, 400 Requests/Second | read-user-timeline<br>compose-post<br>read-home-timeline | 128 | 1.(HomeTimelineServiceProcessor, dispatchCall)<br><br>2.(MediaServiceProcessor, dispatchCall) |
| Ramp performance problem- Constant Stress Load | ● 5 intervals of **increasingly** putting heavy load and then gathering traces in base load<br>Base load configuration:<br>1 Thread, 1 Connections, 10 Seconds, 5 Requests/Second<br><br>● Base to max Heavy load configuration:<br>200 Thread, 200 Connections, 120 Seconds, 400 Requests/Second | read-user-timeline<br>compose-post<br>read-home-timeline | 128 | 1.(HomeTimelineServiceProcessor, dispatchCall)<br><br>2.(MediaServiceProcessor, dispatchCall) |

Constant heavy load and Increasingly Heavier load had the same results in our experiments!

# Experiment 4: Traffic Jam performance problem- Time Window Strategy

There is an occurence of Traffic Jam, if a significant difference between response time of all load tests is observed

1. For each path/func a pairwise comparison of the response time series of neighbouring experiments is conducted
2. For this comparison response time sets for each experiment is gathered and bootstrapped to be normally distributed. Then a t-test is applied to compare
3. If comparison of all sets results in significant growth of response time, then Traffic Jam anti-pattern is observed



# Experiment Results

| Test case | Load intensity | Services tested | # of test functions | (Service, Function) ranked as problematic |
|---|---|---|---|---|
| Traffic Jam performance problem | Put heavy Load 1:<br>20 Thread, 20 Connections<br>120 Seconds, 20 Requests/Second<br><br>Put heavy Load 2:<br>100 Thread, 100 Connections<br>120 Seconds, 100 Requests/Second<br><br>Put heavy Load 3:<br>200 Thread, 200 Connections<br>120 Seconds, 400 Requests/Second<br><br>Put heavy Load 4:<br>500 Thread, 500 Connections<br>120 Seconds, 2000 Requests/Second | read-user-timeline<br>compose-post<br>read-home-timeline | 239 | 1.(HomeTimelineServiceProcessor, dispatchCall)<br>2.(MediaServiceProcessor, dispatchCall)<br>3.(SocialGraphServiceProcessor, dispatchCall)<br>4.(GetFollowers_result, write)<br>5.(GetFollowers_result, read) |

# Conclusions

- Ranking candidate functions to trace in more detail, based on the provided tracing goals
- Increased efficiency of tracing by focusing on specific tracing goals