

# An Automated, Cross-Layer Instrumentation Framework for Diagnosing Performance Problems in Distributed Applications



By **Emre Ates**<sup>1</sup>, Lily Sturmman<sup>2</sup>, Mert Toslali<sup>1</sup>,  
Orran Krieger<sup>1</sup>, Richard Megginson<sup>2</sup>,  
Ayse K. Coskun<sup>1</sup>, and Raja Sambasivan<sup>3</sup>

<sup>1</sup>Boston University; <sup>2</sup>RedHat, Inc.; <sup>3</sup>Tufts University

ACM Symposium on Cloud Computing  
November 21, 2019, Santa Cruz, CA

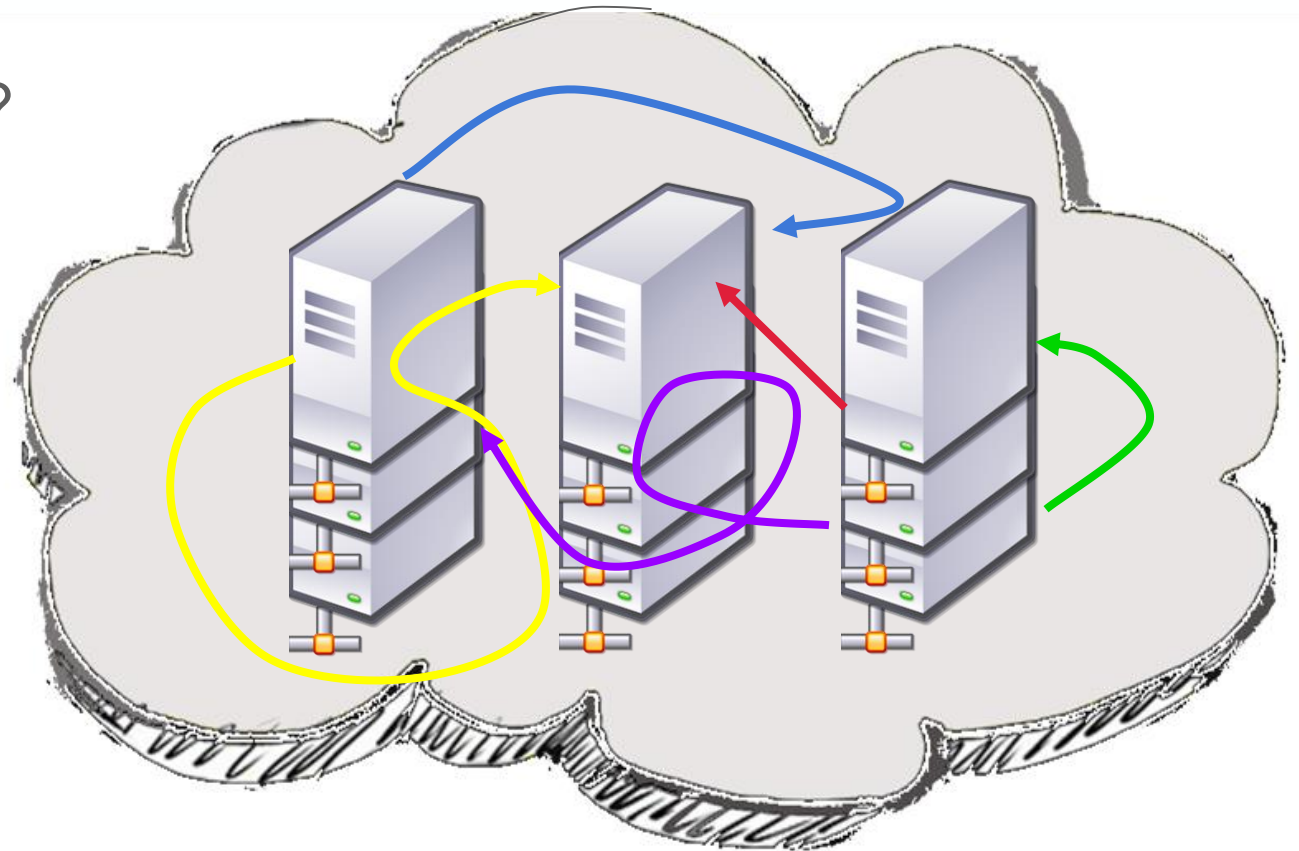


# Debugging Distributed Systems

Challenging: Where is the problem?

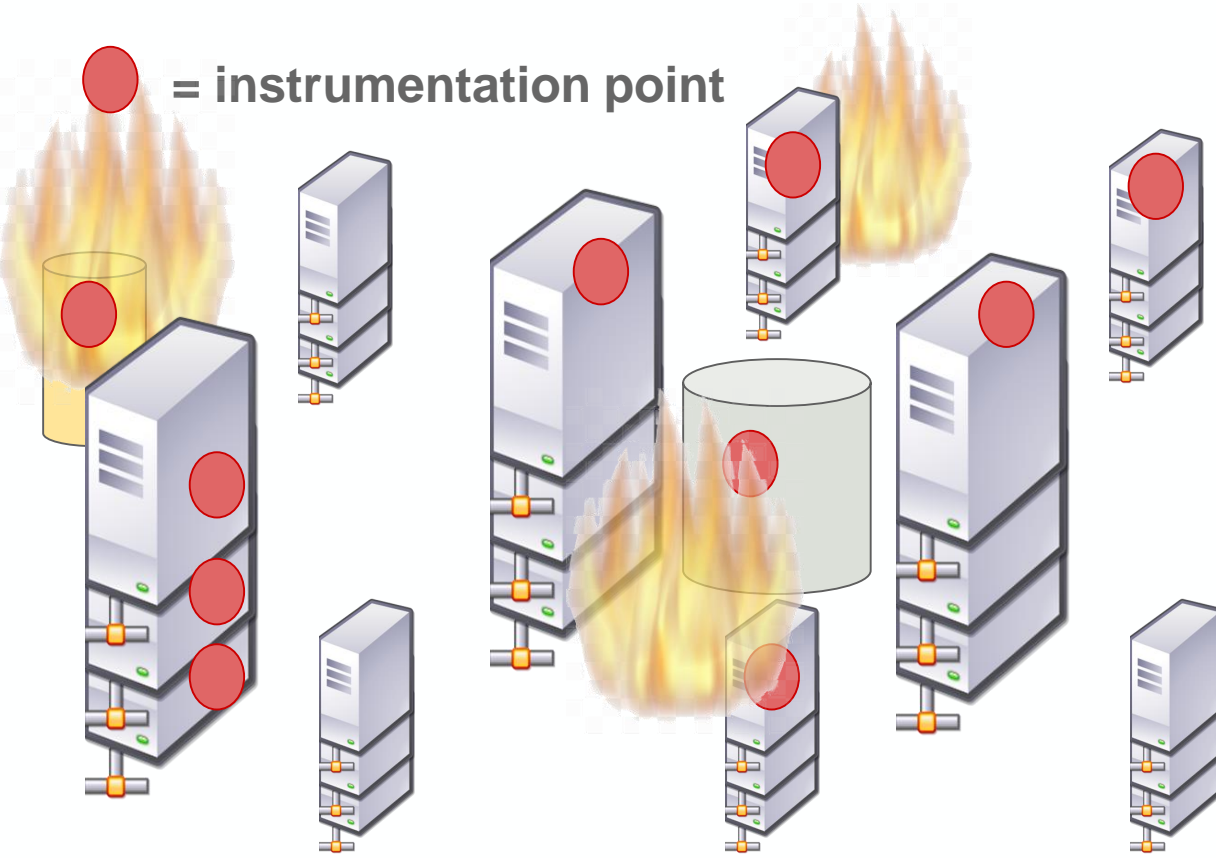
It could be in:

- One of many components
- One of several stack levels
  - VM vs. hypervisor
  - Application vs. kernel
- Inter-component interactions



# Today's Debugging Methods

● = instrumentation point



## Instrumentation data

```

1>kjP~oiu._io&UKE@ Qds= Sask,jlk<utiud~zxs2&33r/vft@ io328.bLPyh-09,
vs$mmas# wi934<.?TY# * H! b,76<bkoKJ~d%*klj. (eruTG! S23)_Ulgh^798
>6KT$sfj<riguTRXH'15~$fg$ kd%1u@ (guj49_) "skdf* M\ 5Nb# vuh45~xn
j>kjP&oiu._ioRE$ Qds= Sask,jlkhiu^d~zxs231 3r/vfti# o3&8.bLP2@y^h-09
$%^@1wsas! wi934<.?TYHb,76<by$ 7kod%*klj. eruT# GS*3)_Ulg^gh^798
jif<<riguTR^X~H'15#$fg! kd# u(gu@^j49_) "skdfNbvuh45~xnz87`zdg%LK
oAo@7!{48tg!}6eSW!_sSTj: YA+Fd/dsf%~66#?hh*88$ 45~xnz^87`zdg* L%
_ioO%R4 $E*Qds= Sask,jlk^iud~zxs233r/vfti# o3^%8.bl@Pyh-0923!";oijh
vsa~swi934<.?TYH^ b,76<by7~kod%*klj. eruT# GS23)_Ulg^gh^798mn,23
/v& fti# o32^ 8.bLP~yh-0923!";oijht)yi# oHg7 &569=sDERTg^ tg9*&ru~y!
iT# GSff$23)_Ulgh^798mn,2334hg!k! gj:"p[o]}o@o.yg&Z# X&*q! o>6KT$:
@h45~xnz87`zdgL~K4$ 9%8P?...JHY@|"ser3^ 2Z`~54drts%h.<ut(jRus) 72
#?hh*8*45~xnz87`zdg^ LK4> 98P?...JH*Yg|"ser32Z`~54dr^ t$%h.d>kjPoi
28.blPh-0923!";oij^ ht)yi# oHg569=sDEK RT$ gt^g9*&! ruyf$%^@1wsas< wi934<.?TYHb,76<by7ko^ %*klj. eruT# GS2h 3)_
j! gh^798^ ddn,2334h$ glkj:"p[o]}oo..y*g&Z# X&qo>6d>kjPoiu._ioOR! E@Qds= Sask,jlk<ut& iud~zxs23! 3r/vft@io$ 328.bLP!
0923!";oij% ht)y~iuo$ Hg59=sD! RTg6 t$ g9*&ruf$%^@1!lS w& %s^ # wi934<.?TY# Hb,76<bkoKJ~d%*klj. (eruT^G! S23)_Ul
^798mn,23^34# hMN>=glk~gj:"p[o]}ooy# g&% Z@&qo>6K7 T$sfj<rigu! TR@XH'15#$fg! kd1u (gu*j49_) "skdfM\ 5Nb# vuh45
z87`zdgLK4> 98P?...JHYg|"ser32Z`~54drts%h.d>kjPo"iu._ioOR~E$ Qds= Sask,jlkhiud~zxs`23~3r/vfti# o3&8.bl$ Pyh-0923!";
^ht)yi# oHg569=sD&^RTg$ tg9*&ruf$%^@1wsas~wi934<.?TY! Hb,76<by7k*od%*klj. eruT# GS23)_U!ggh^798mn,23^34h
jz:"p[o]}oo..yg&Z# X&qo>6KT$sfj<<riguTRS XH'15#$fgkd# u(gu% j49_) "skdfNb$ vuh45~xnz87`d67g%LK4~8P?...JYg|"se^r3yy,
~54drts%h.<ut(jRus) 72+joA*o%o7!{48tg!}6eSW!_sSTj: YA+Fd/dsf%& 66! #?hh^*88@45~xnz87`zdgLK4> 9:P?...JH{ g|"sr32Z`~
54drts%~h.d>kjPoiu._ioORE^Qds= Sask,jlk^iud~zxs$ 233r/vfti# o328.blPyh-0923!";oijh^t)yi# *oHg& 569=sDER@Tg% tg9*
uyf$%^@1wsa$ sw@9&34<.?TY~Hb,76<by! 7kod%*klj. eruT# GS23)_Ulgh^798mn,2337 4h^ glk& gj:"p[o]}oo..yg&Z# X&
>>6khiud~zxs2^33r/vfti# o328.bl@Pyh-0923!";oijht)8* yiu# oHg569=s~DE&@R~Tgtg9*&ruf$%^@1wsa@^ swi9@34<.?TYHb
5<by7ko*d%*klj. eruT# GSff 23)_Ulgh^798^mn,23% 34! h% glkj:"p[o]}o^ o..yg&Z# X&qo>6%K^$sfj<<riguTRXH'15#$fg$ kc
u(guj49_) "skdf^@Nb! vu% h45~xnz87`zdgLK^ @498P?...JHYg|"se^r32Z`~54drts%h.<ut(jRus) 72+joAoo7!{4~8tg!}6eSW!_sST
: YA+Fd/d$ sf^ %66! #?hh*8845~xnz87`zdgLK4> 98P?...JHYg|"ser^ 32Z`~54drts%h.d>kjPoiu._ioOR& E! Qds= Sask,jlk^hiud~
s233r/vfti# o328.b^Ph-0923!";oij%ht)yi# oHg569=sD~RTg7& @tg9*&r^ uyf$%^@1wsas< wi9$ *34<.?TYHb,7~6<by7kod%

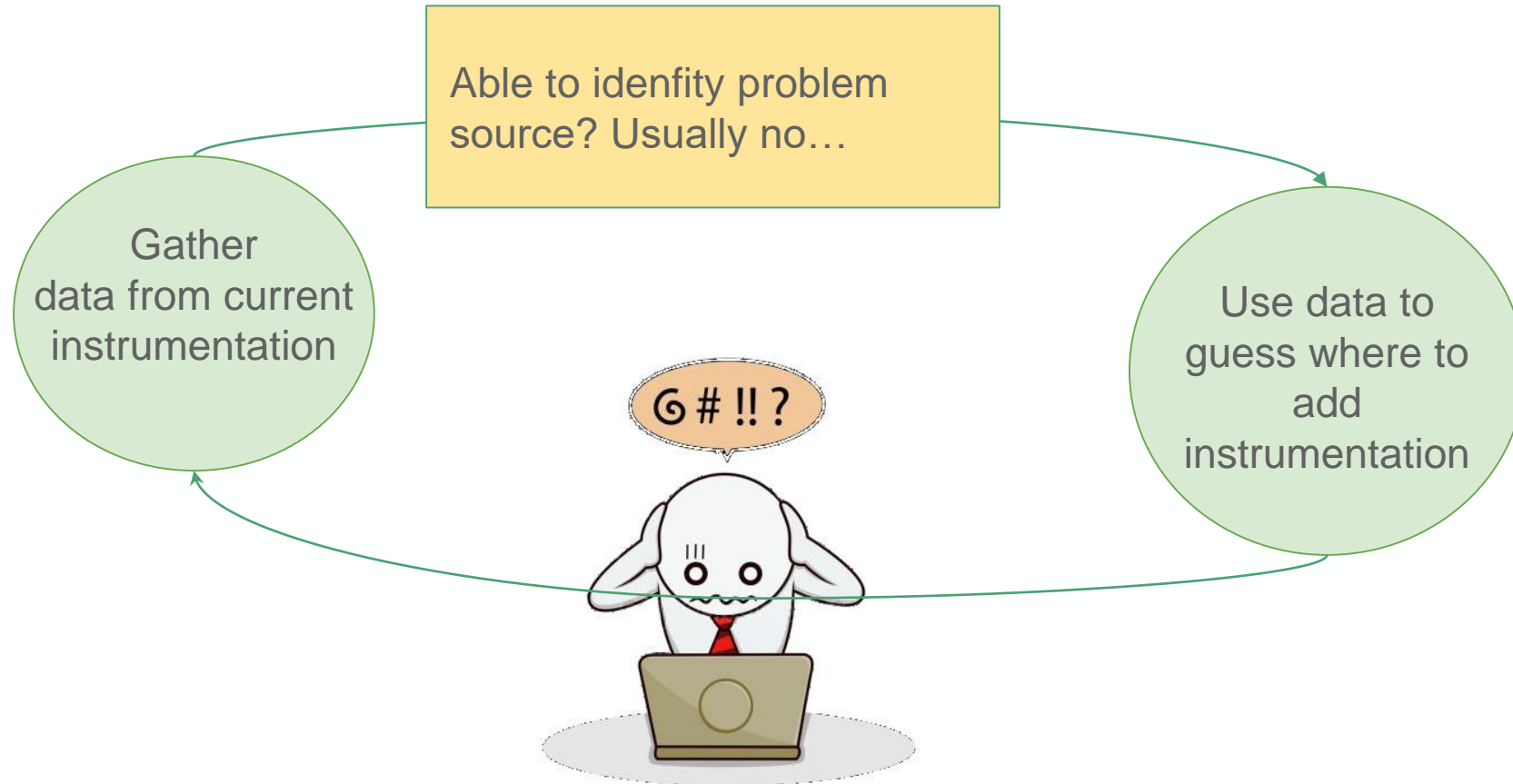
```



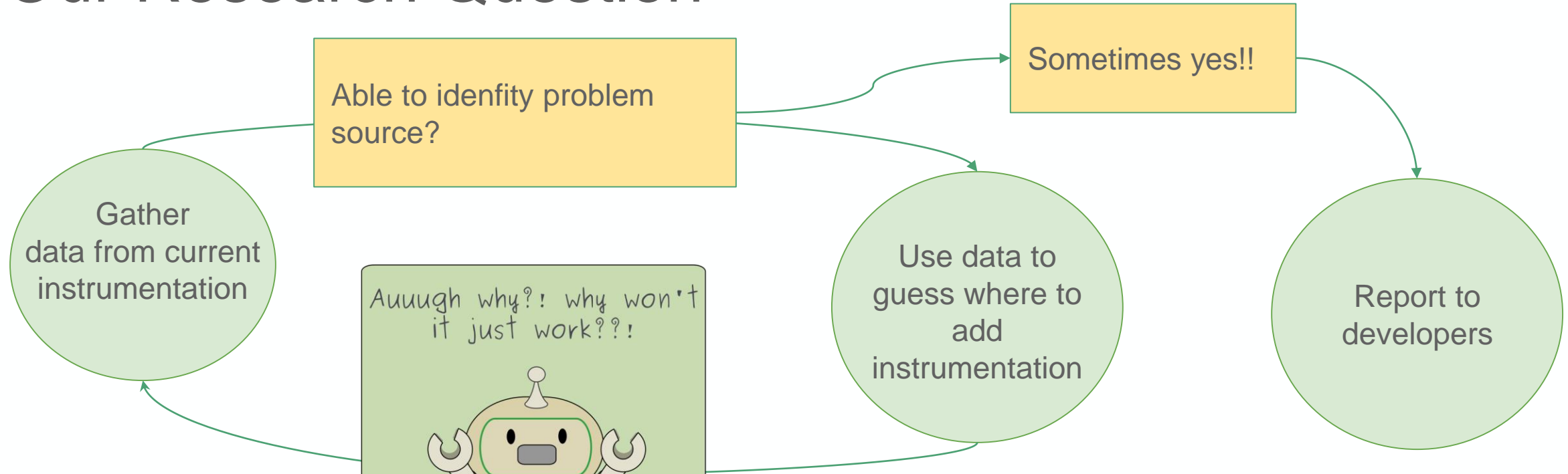
**Different** problems benefit from **different** instrumentation points.

You can't instrument everything: too much overhead, too much data.

# Today's Debugging Cycle



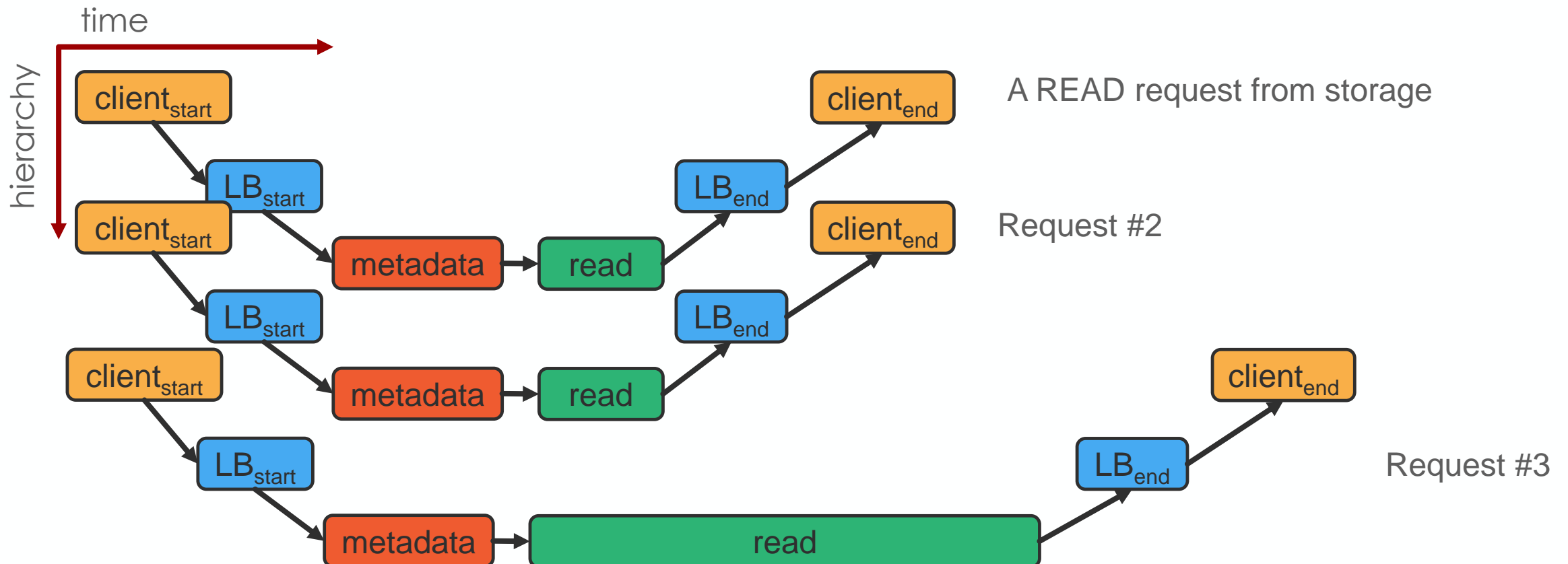
# Our Research Question



Can we create a **continuously-running** instrumentation framework for production distributed systems that will **automatically explore instrumentation choices** across stack-layers for a newly-observed performance problem?

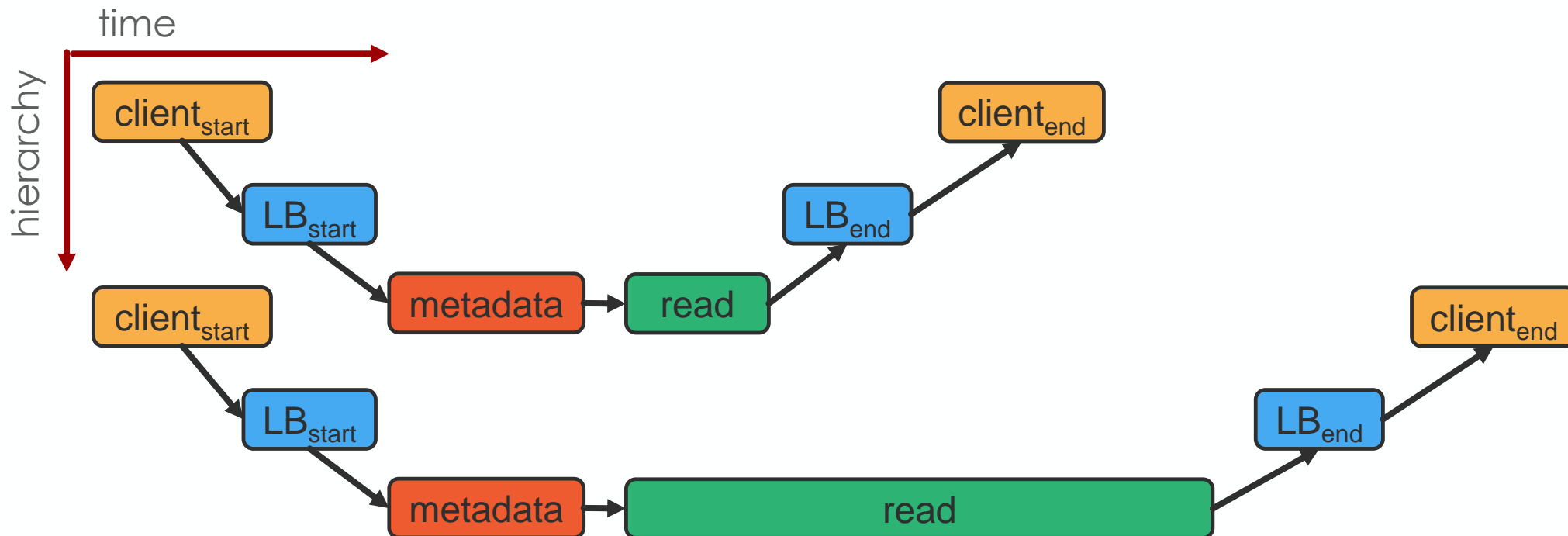
# Key insight: Performance variation indicates where to instrument

- If requests that are expected to perform similarly do not:
  - There is something unknown about their workflows, which could represent performance problems
  - Localizing source of variation gives insight into where instrumentation is needed.



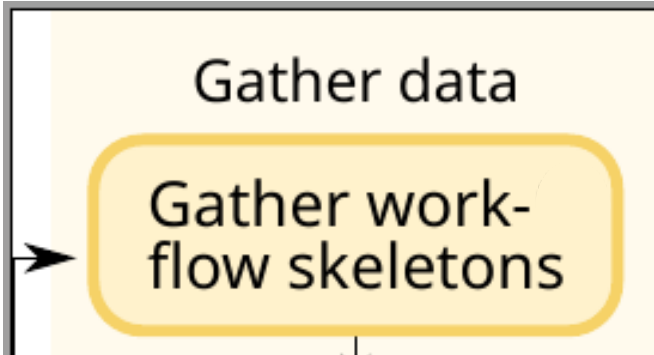
# Key Enabler: Workflow-centric Tracing

- Used to get workflows from running systems
- Works by propagating common context with requests (e.g., request ID)
  - Trace points record important events with context
- Granularity is determined by instrumentation in the system



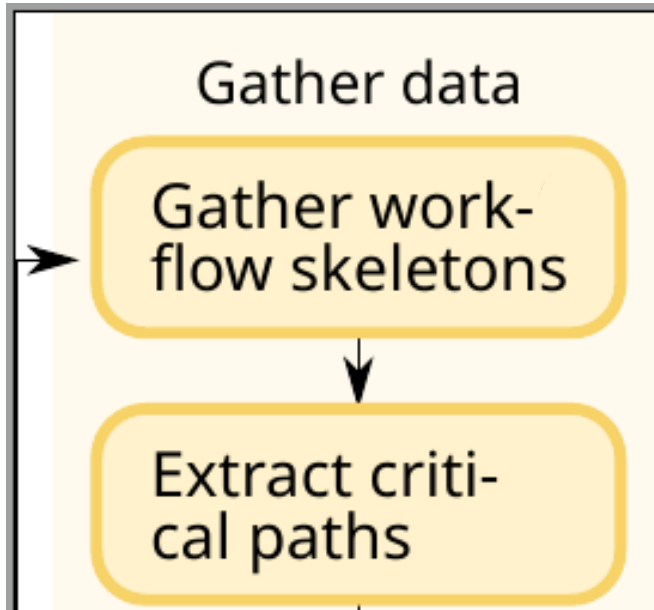


# Vision of Pythia

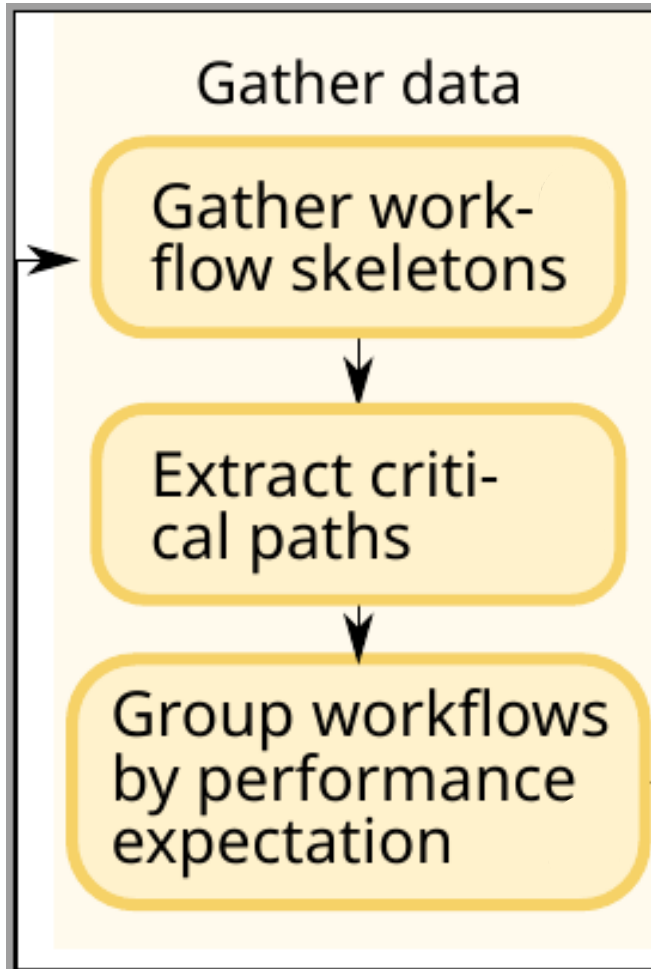




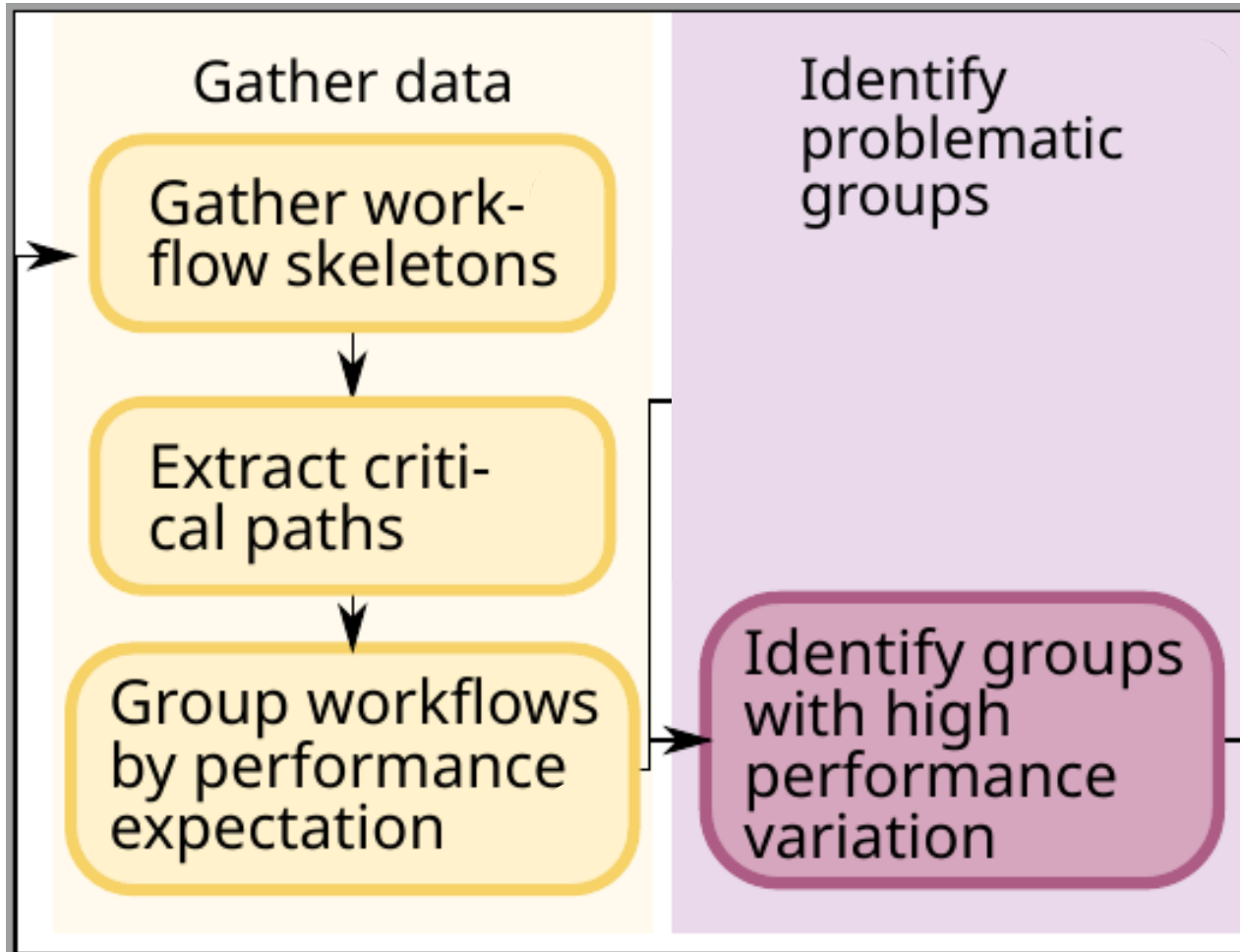
# Vision of Pythia



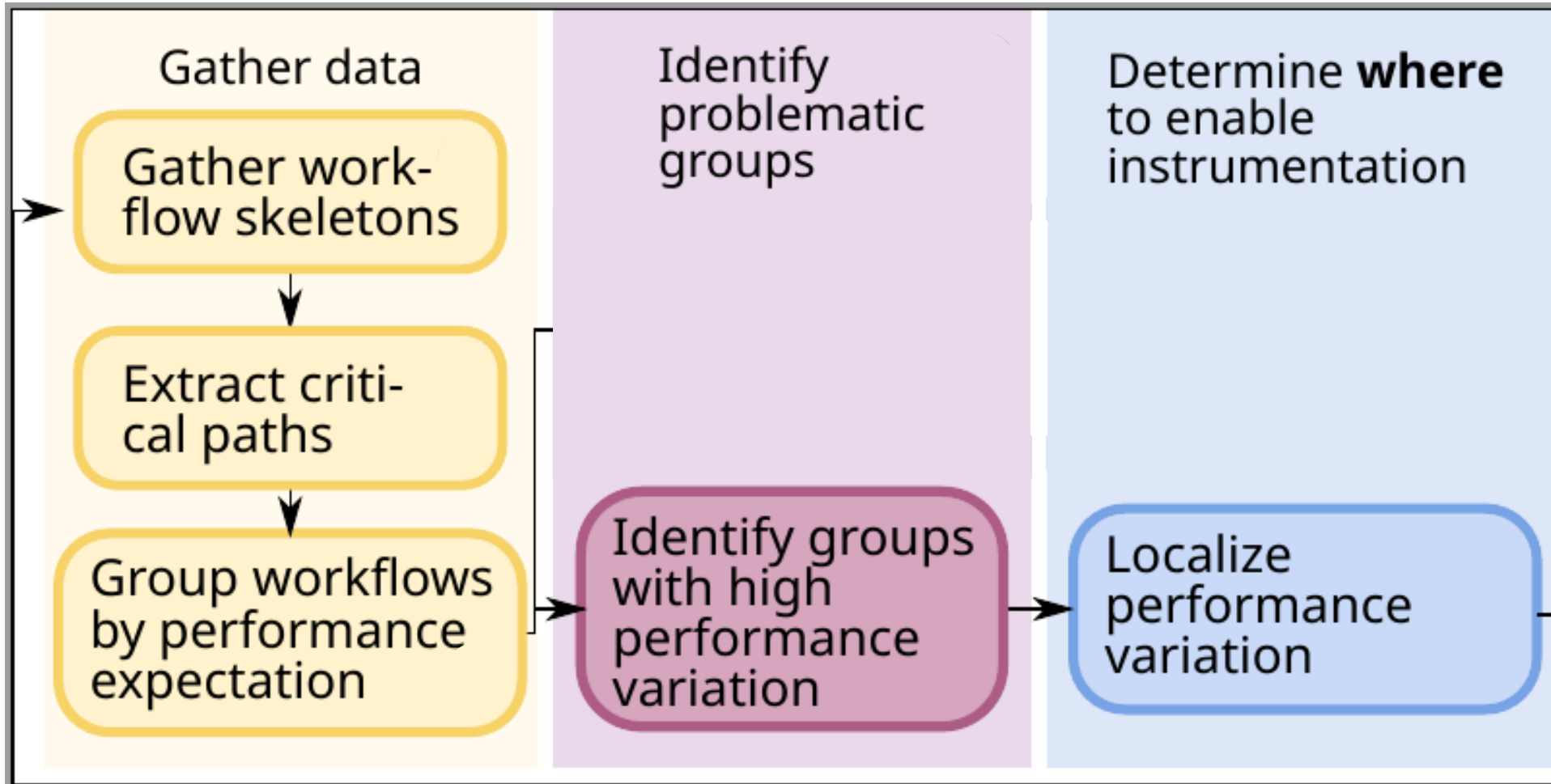
# Vision of Pythia



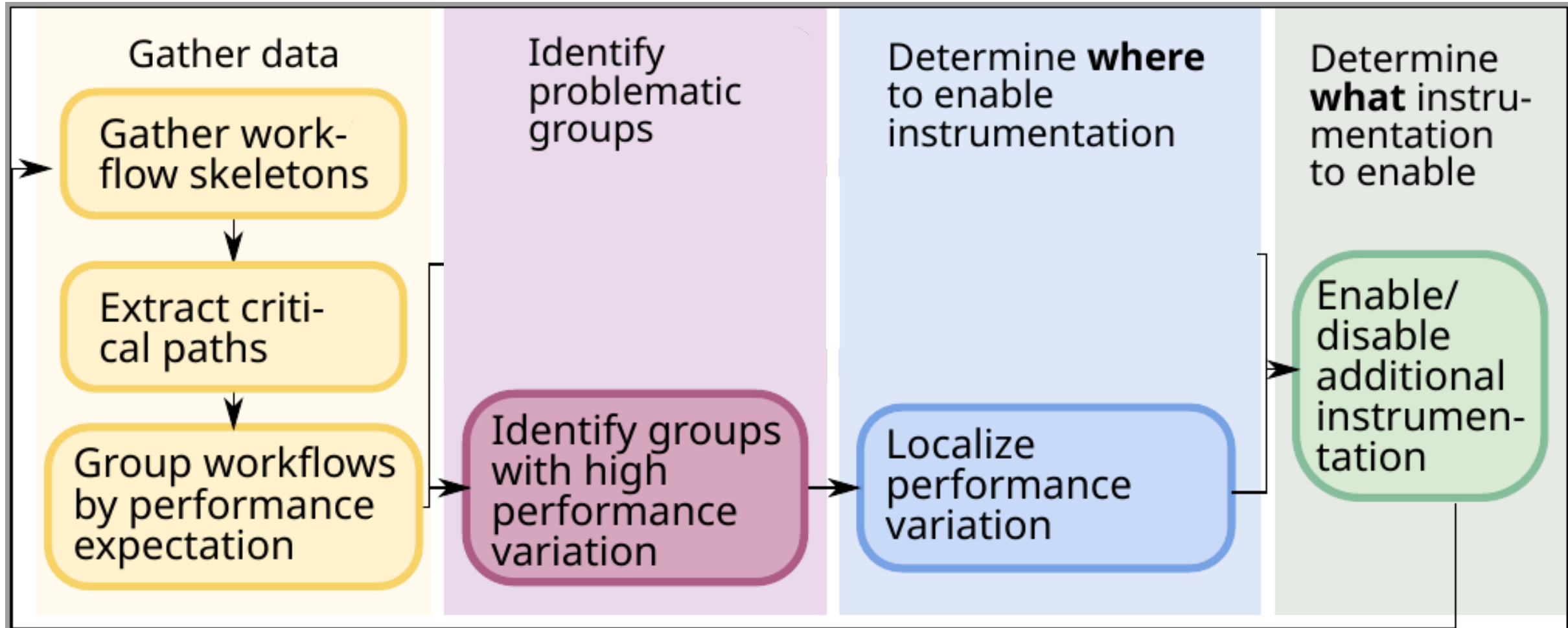
# Vision of Pythia



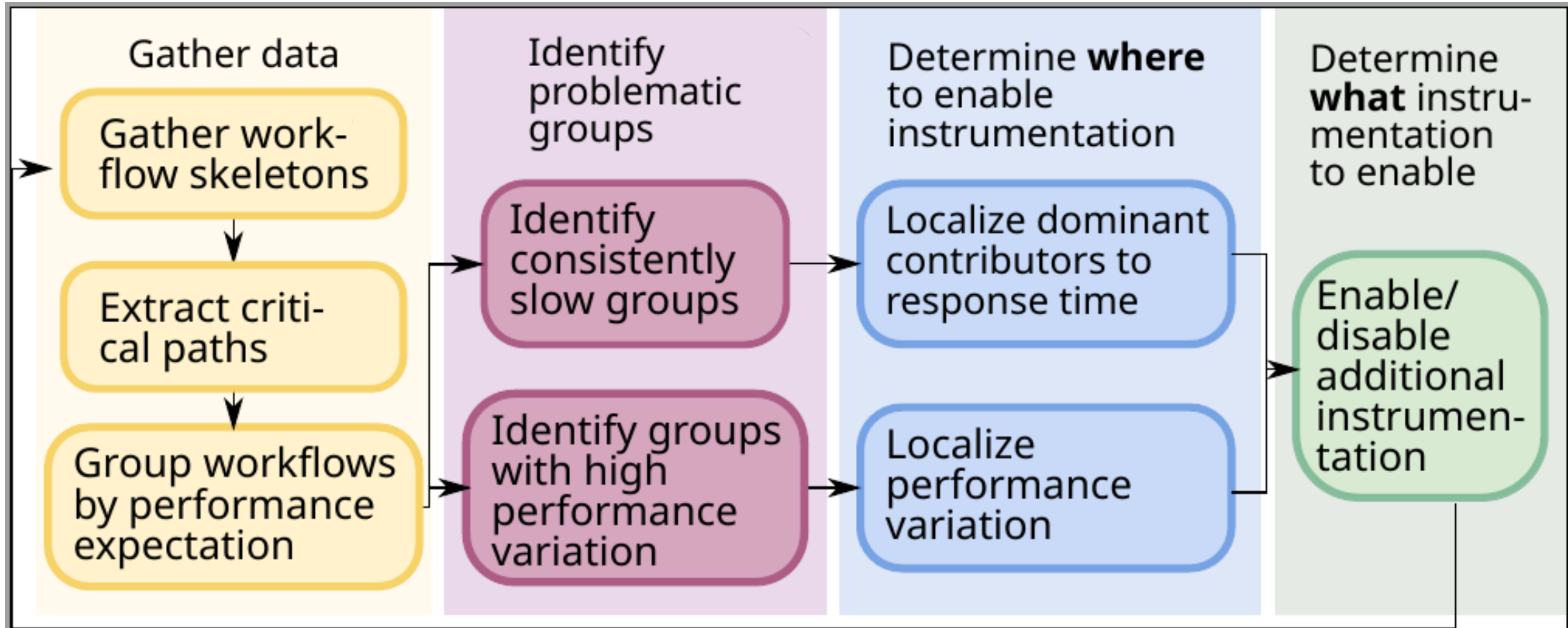
# Vision of Pythia



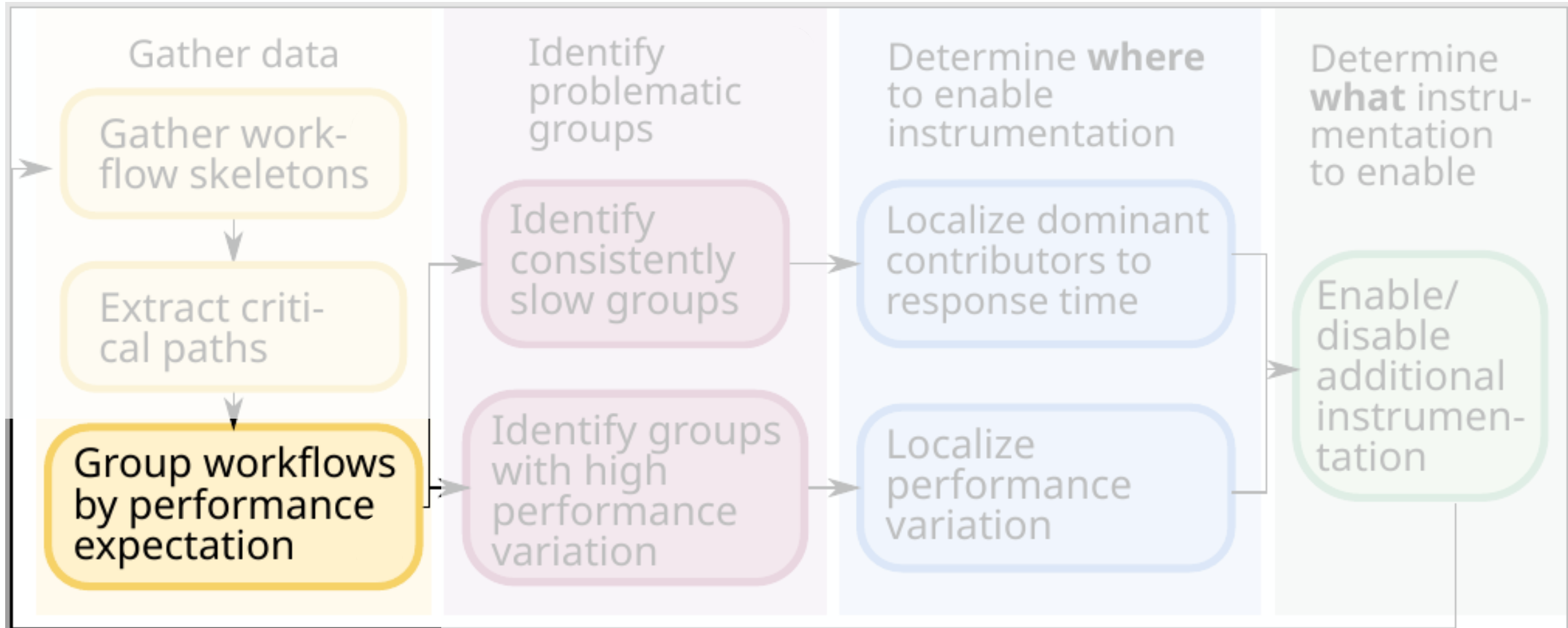
# Vision of Pythia



# Vision of Pythia

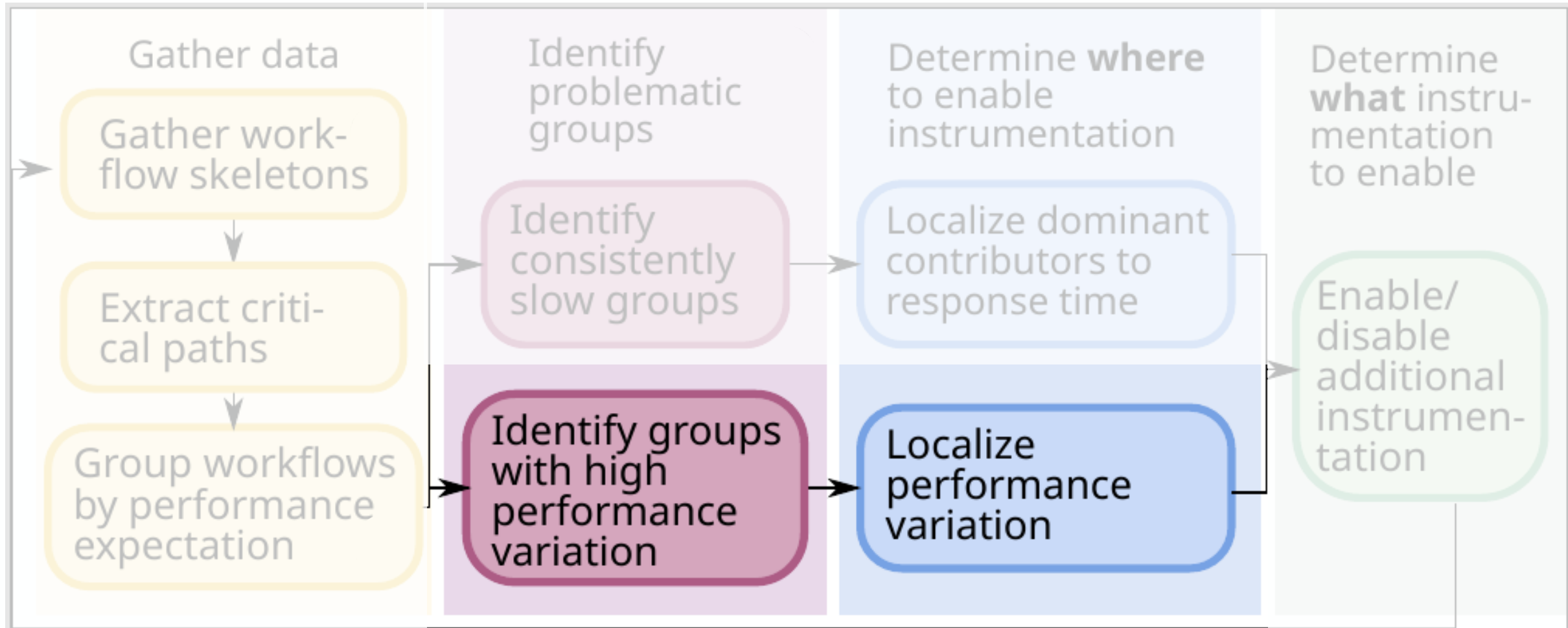


# Vision of Pythia

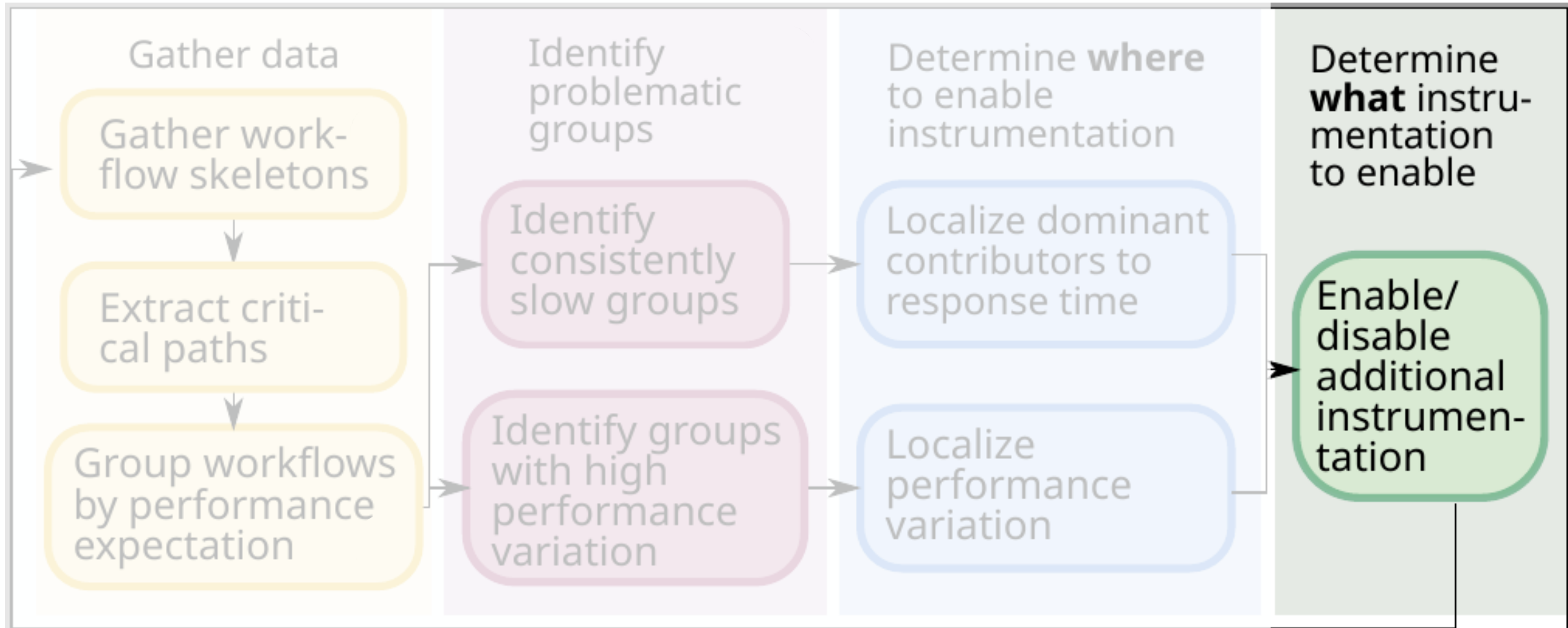




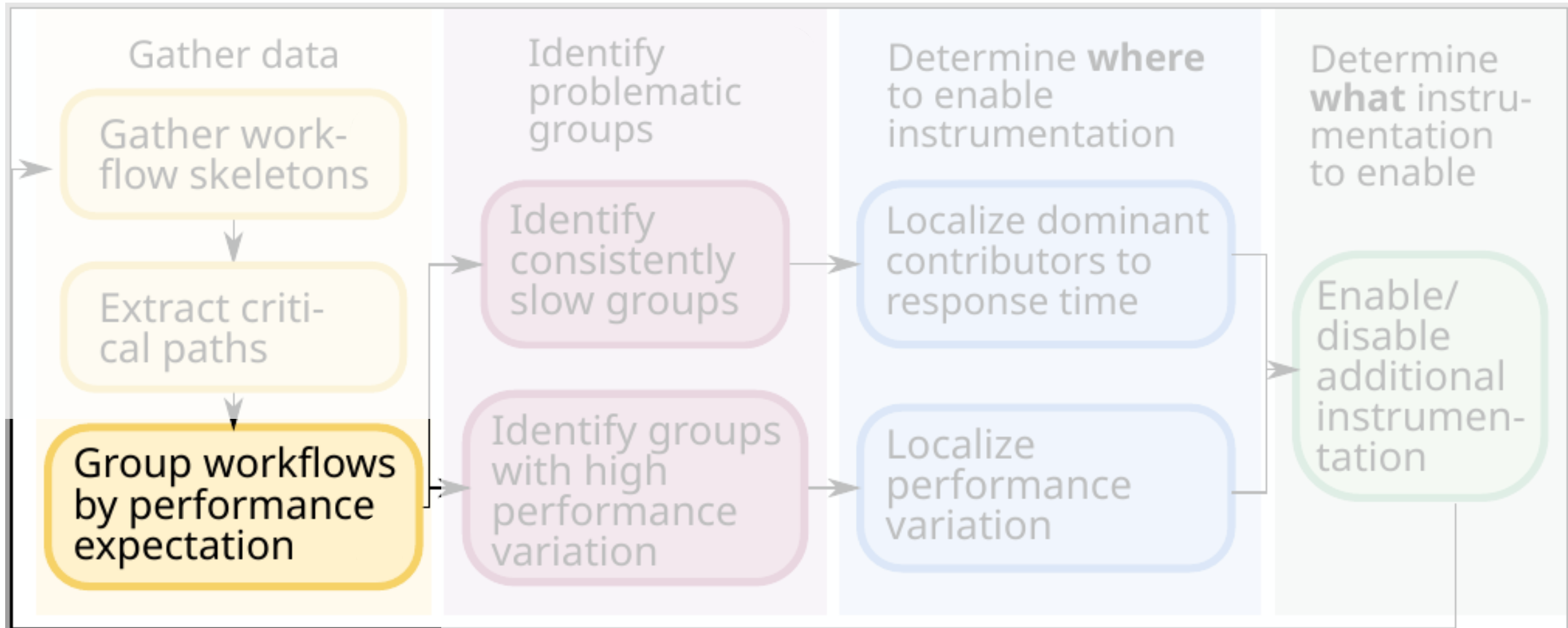
# Vision of Pythia



# Vision of Pythia

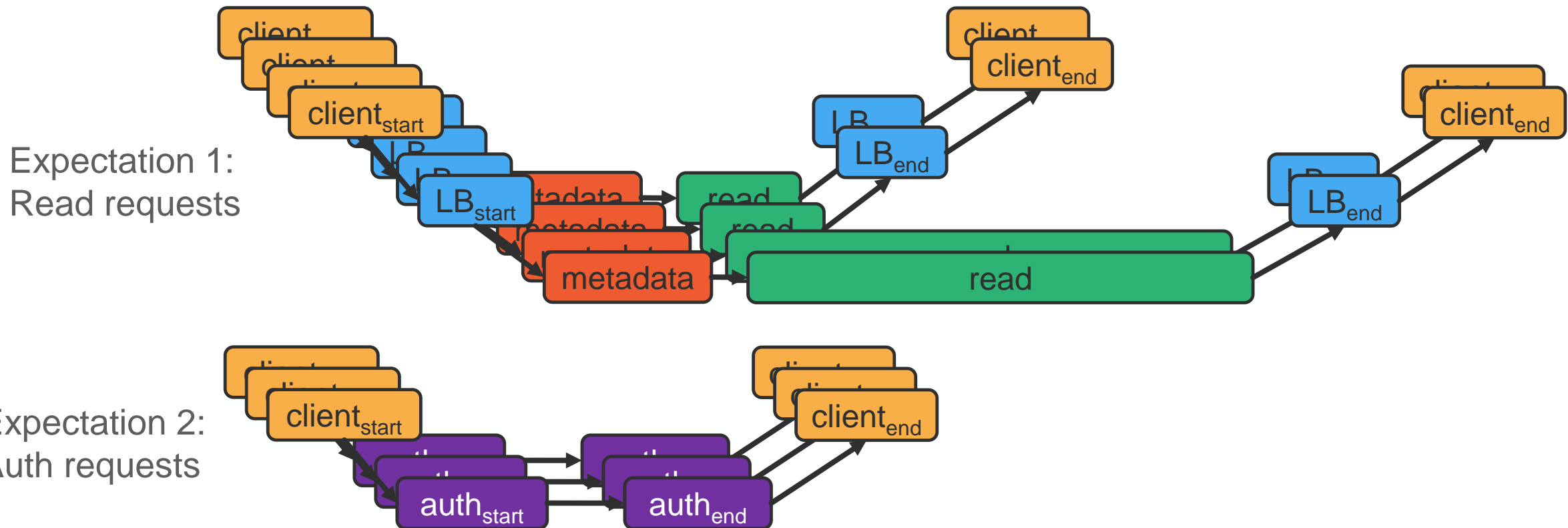


## Challenge 1: Grouping

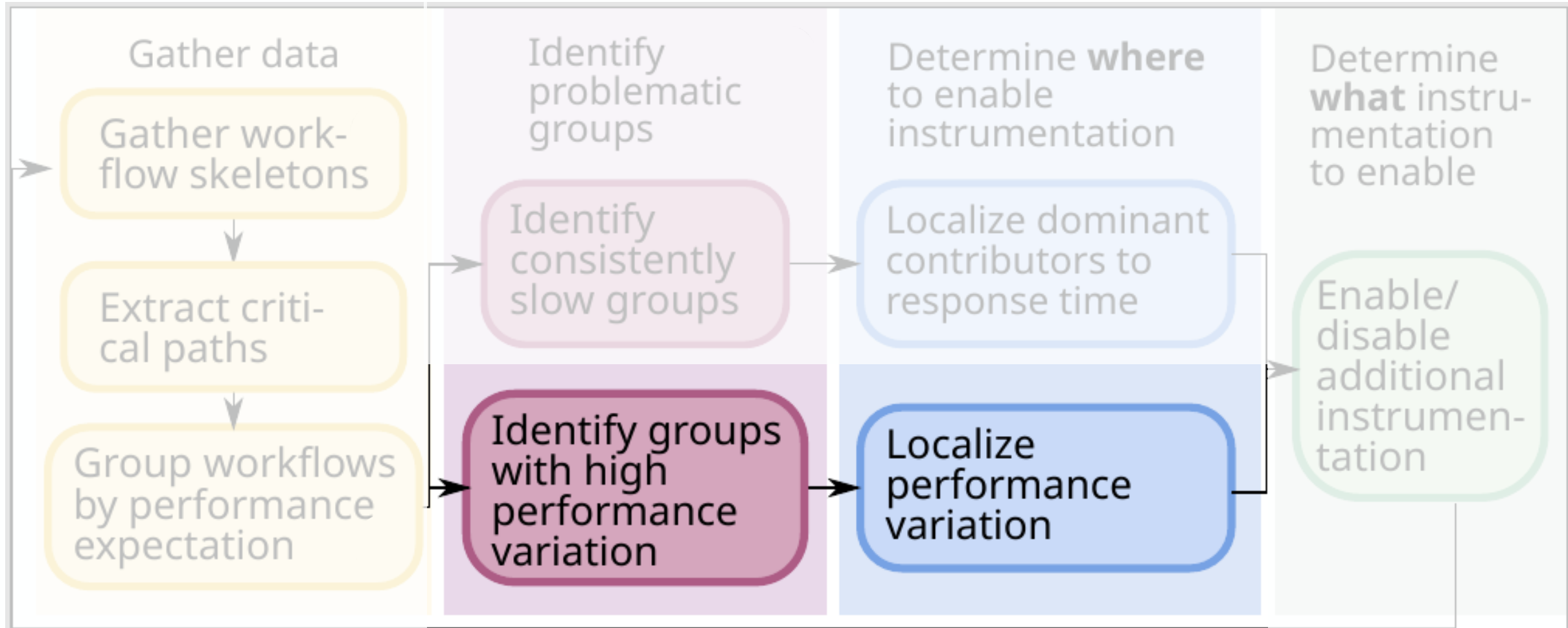


# Which Requests are Expected to Perform Similarly

- Depends on the distributed application being debugged
- Generally applicable: Requests of the same type that access the same services
- Additional app-specific details could be incorporated

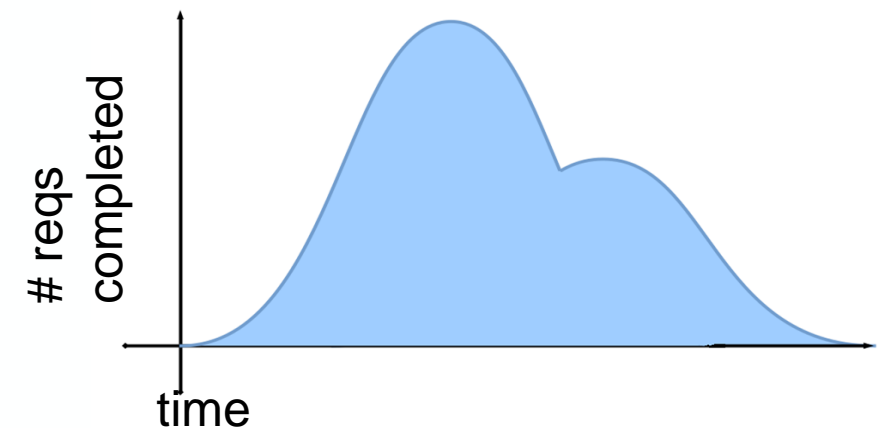
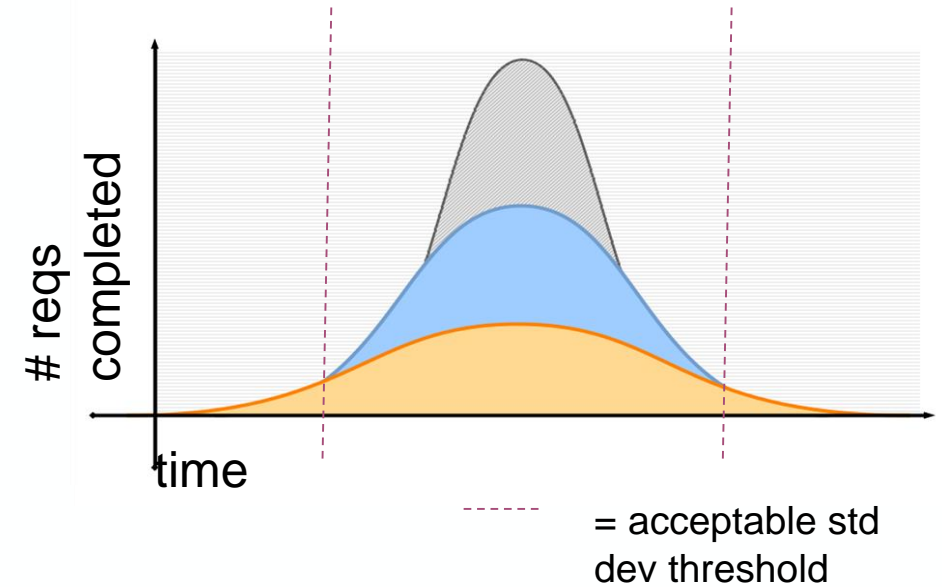


## Challenge 2: Localization

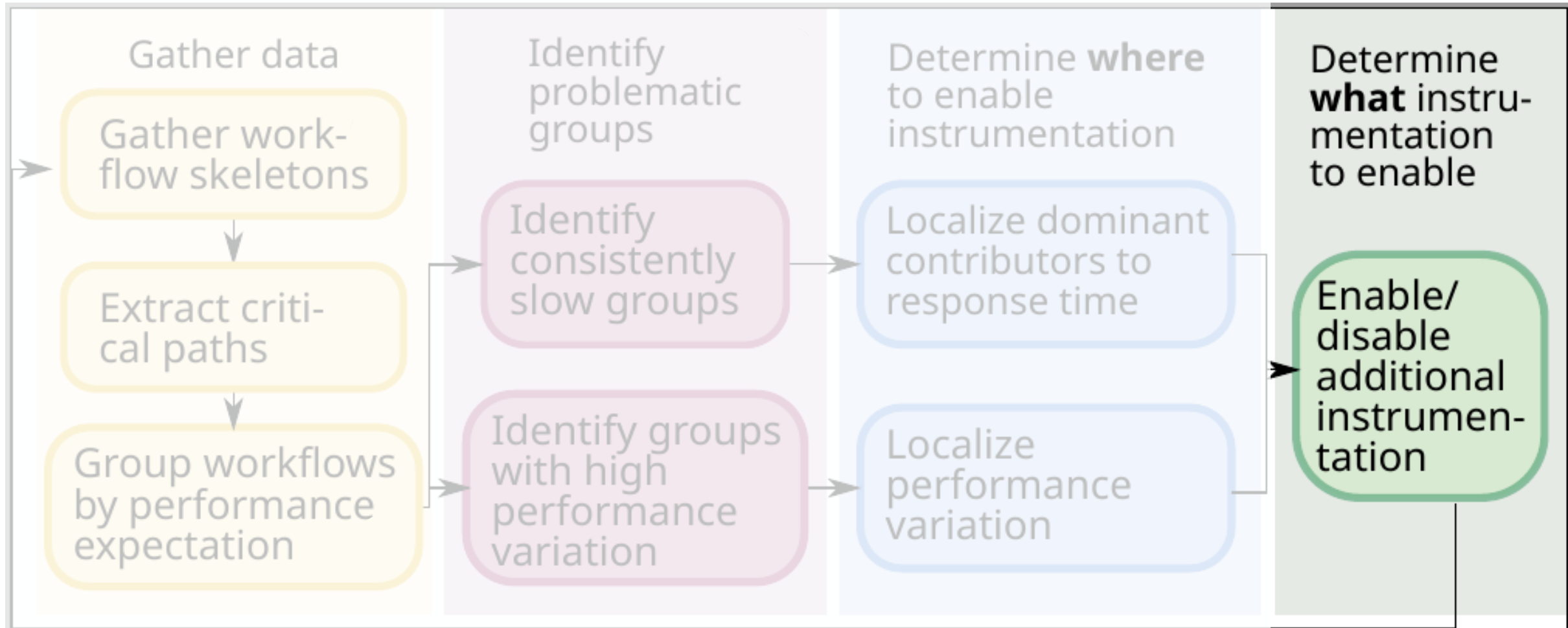


# Localizing Performance Variations

- Order groups and edges within groups.
  - How to quantify performance variation?
- Multiple metrics to measure variation
  - Variance/standard deviation
  - Coefficient of variance (std. / mean)
    - Intuitive
    - Very small mean -> very high CoV
  - Multimodality
    - Multiple modes of operation



## Challenge 3: What to enable





## Search Space

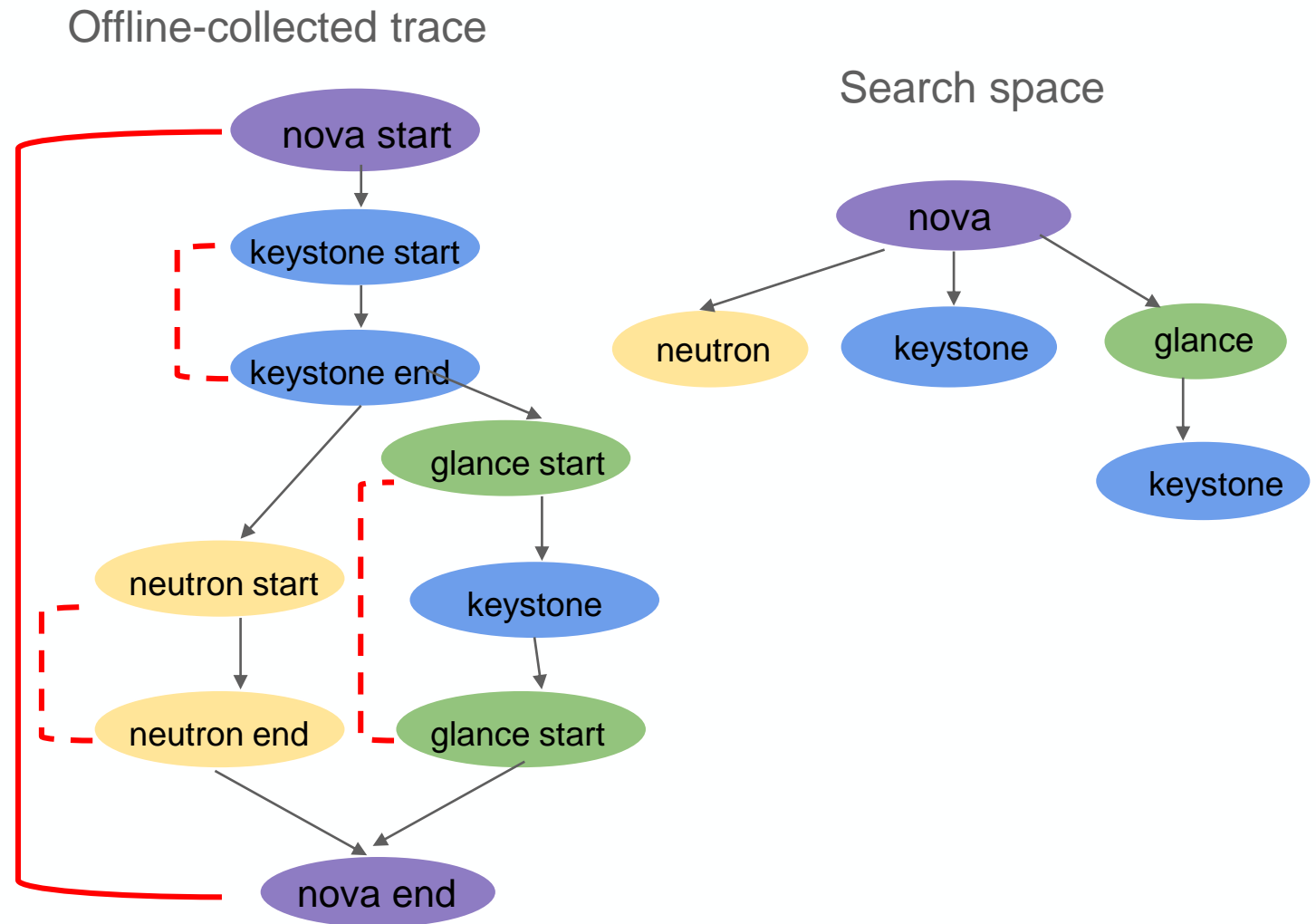
- How to represent all of the instrumentation that Pythia can control?
- How to find relevant next-trace-points after problem is narrowed down?
- Trade-offs:
  - Quick to access
  - Compact
  - Limit spurious instrumentation choices

## Search Strategies

- How to explore the search space?
  - Quickly converge on problems
  - Keep instrumentation overhead low
  - Reduce time-to-solution
- Many possible options
  - Pluggable design

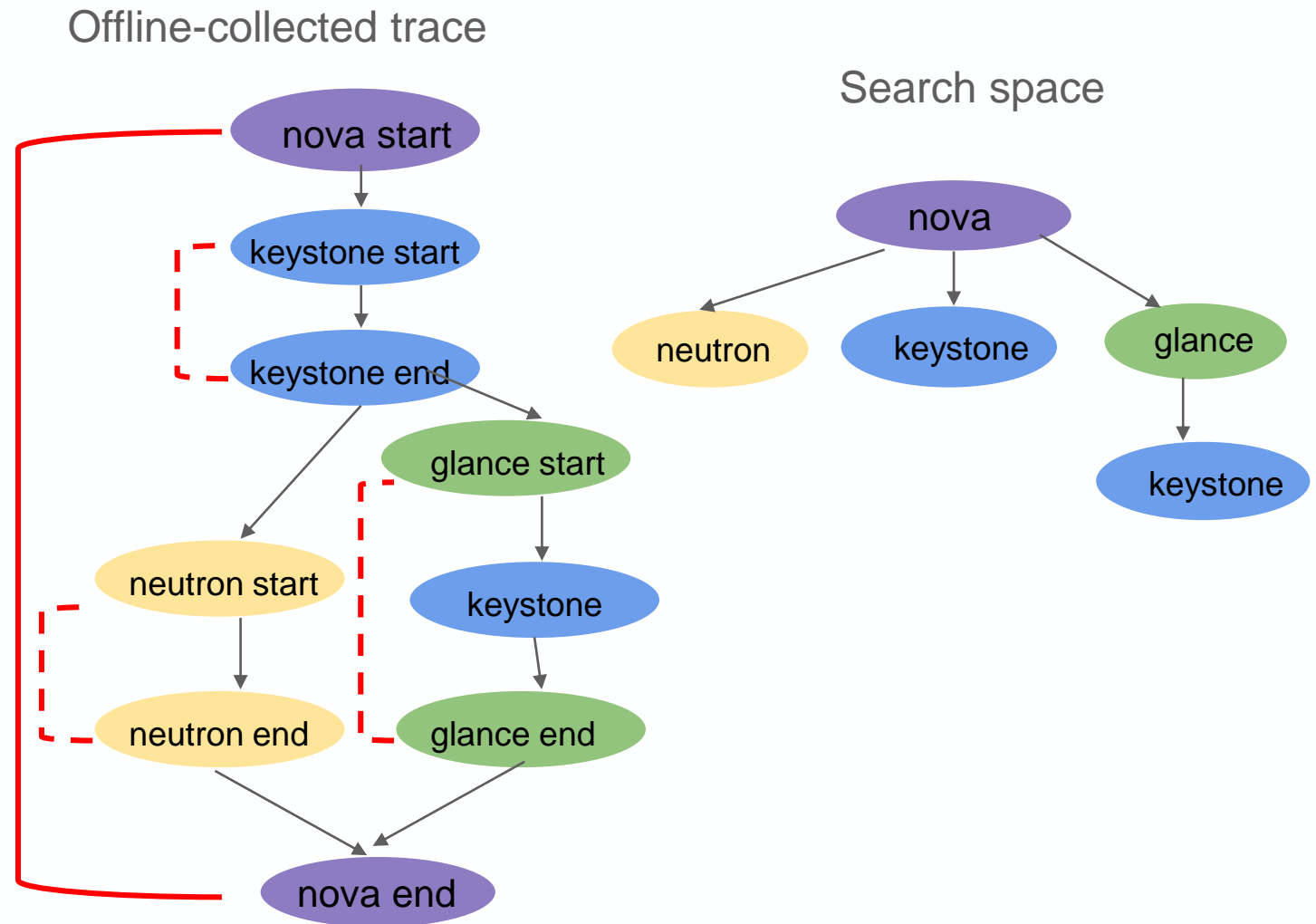
# Search Space: Calling Context Trees

- One node for each calling context i.e., stack trace
- Leverages the hierarchy of distributed system architecture
- Construction: offline profiling
- Trade-offs
  - Quick to access
  - Compact
  - Limit spurious instrumentation choices



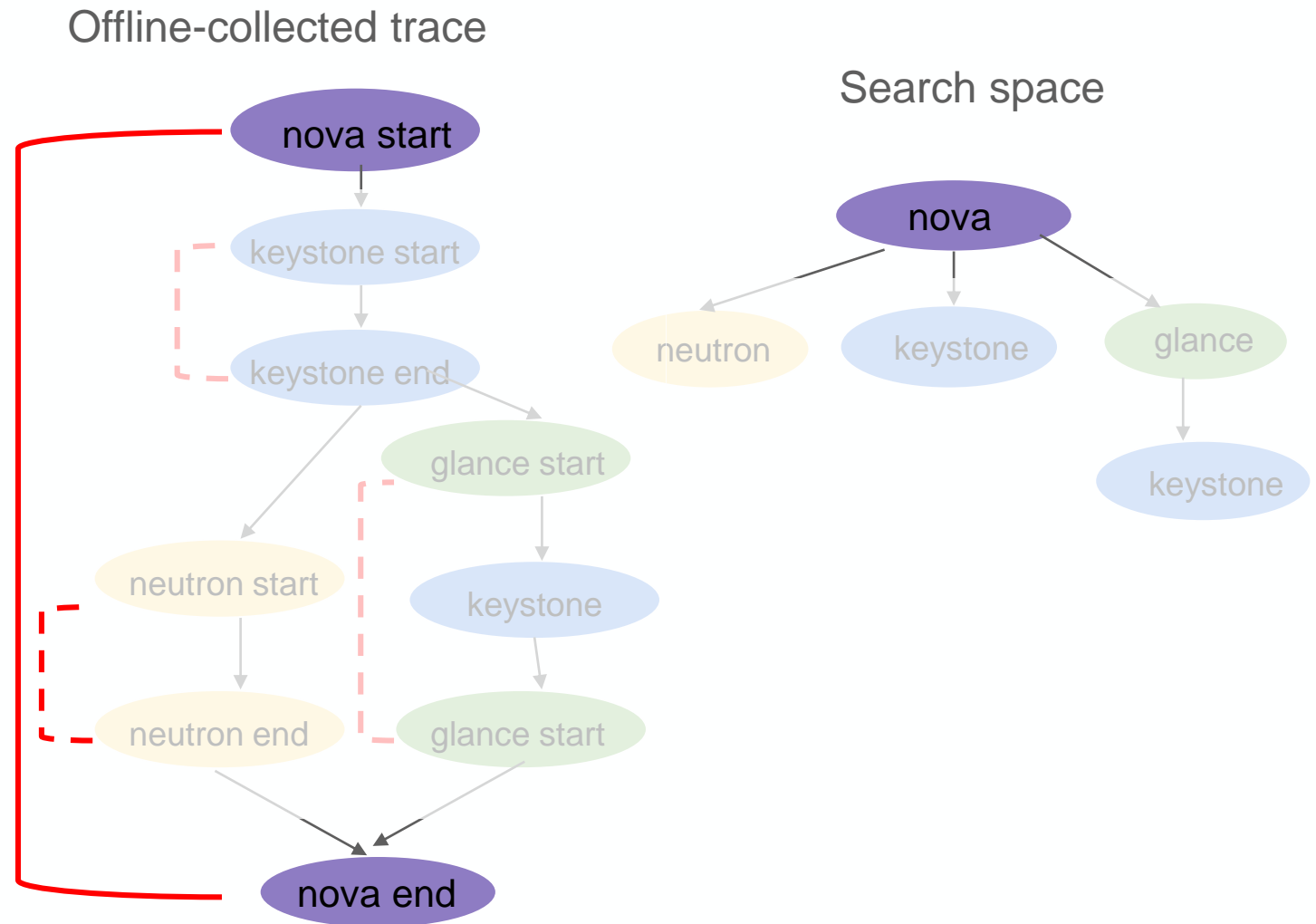
# Search Strategy: Hierarchical Search

- One of many choices
- Search trace point choices top-down
- Very compatible with Calling Context Trees



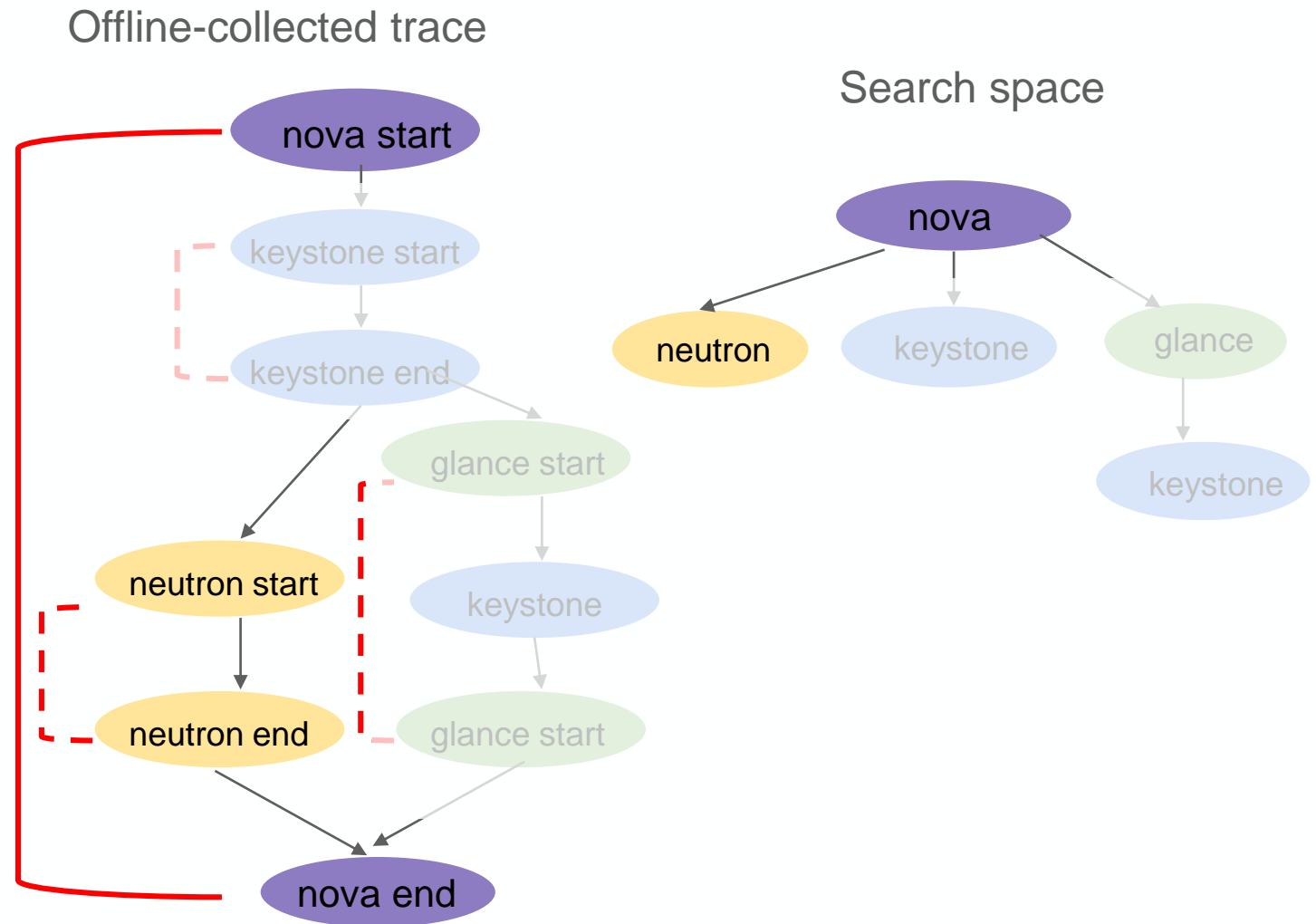
# Search Strategy: Hierarchical Search

- One of many choices
- Search trace point choices top-down
- Very compatible with Calling Context Trees



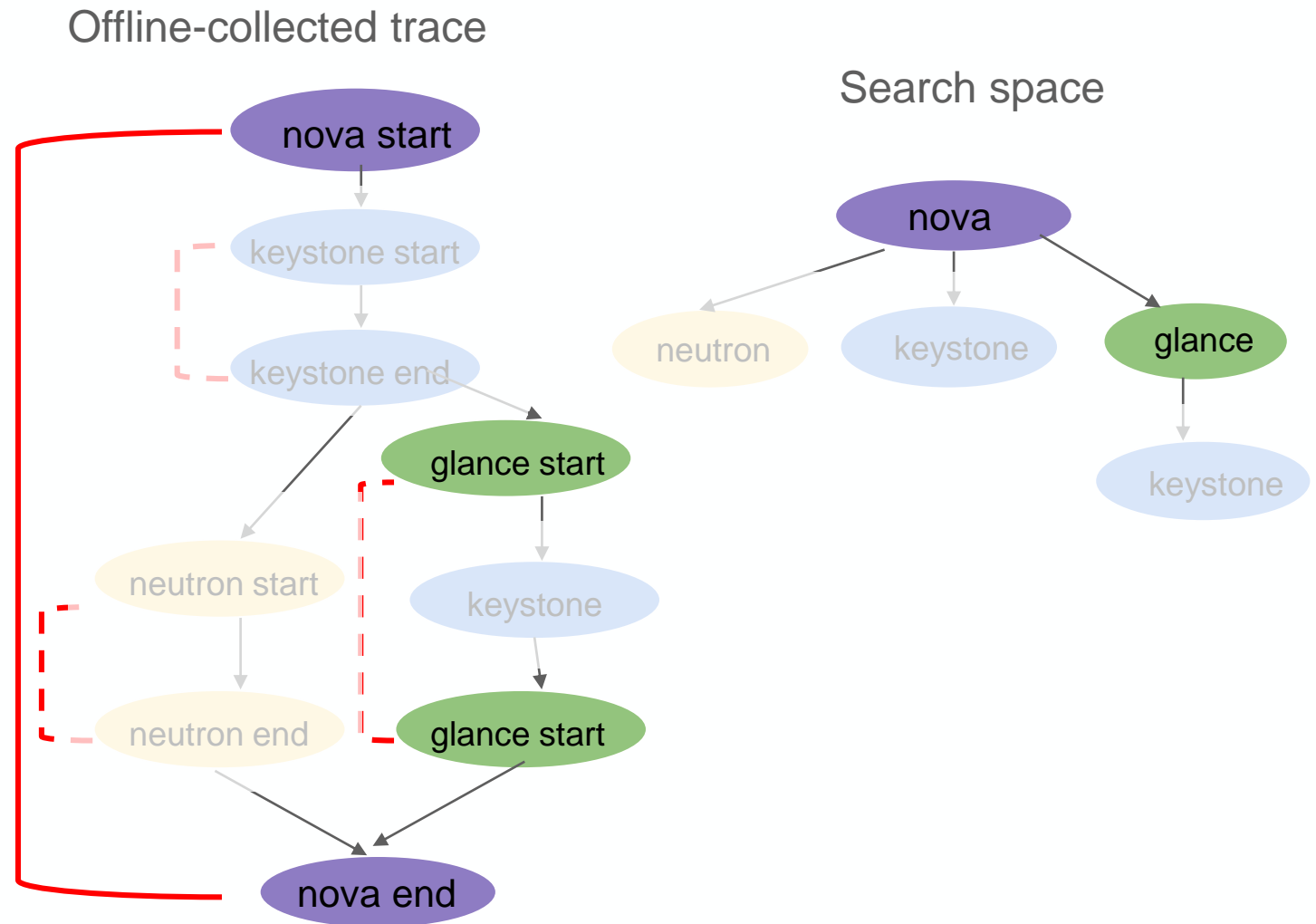
# Search Strategy: Hierarchical Search

- One of many choices
- Search trace point choices top-down
- Very compatible with Calling Context Trees



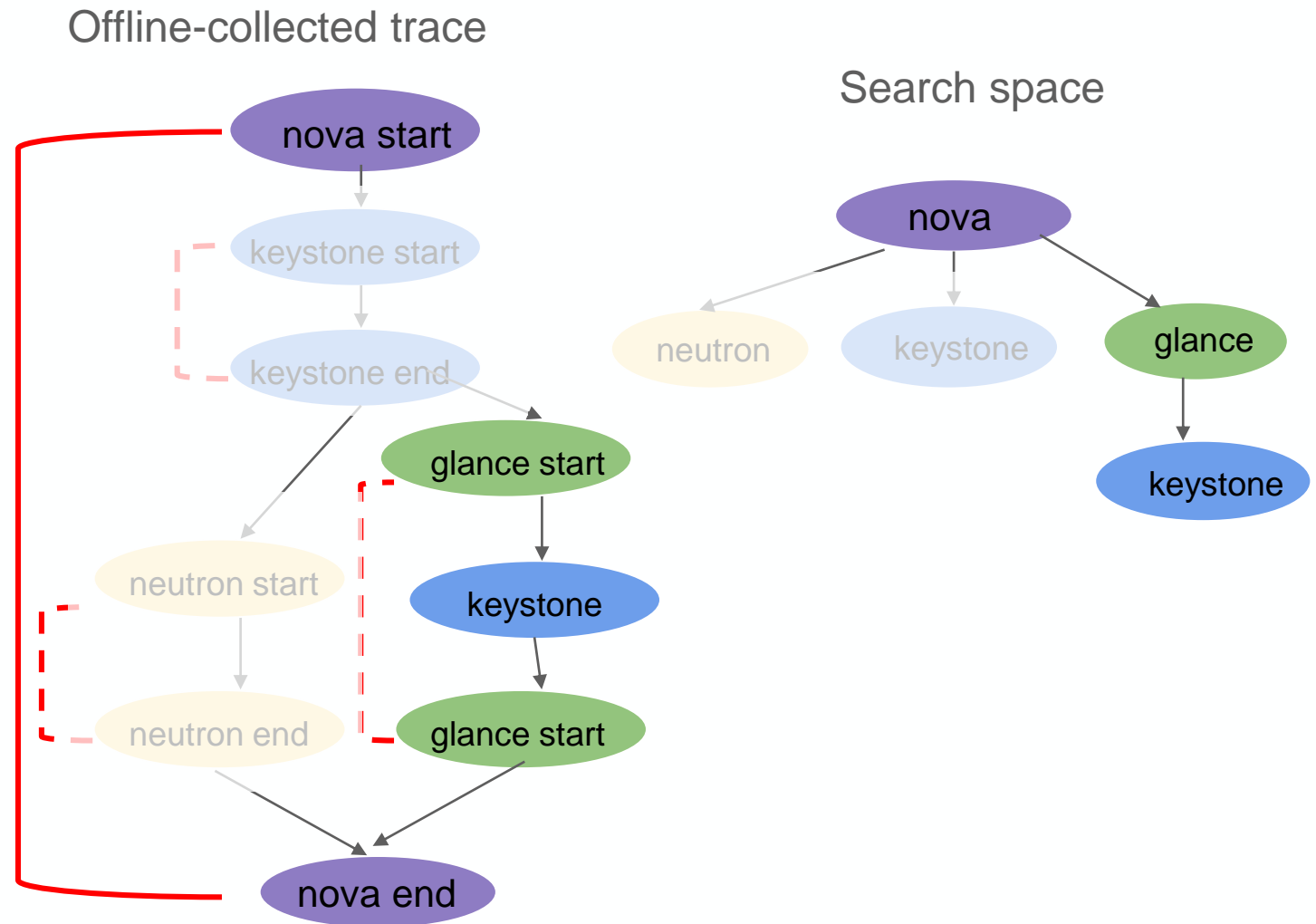
# Search Strategy: Hierarchical Search

- One of many choices
- Search trace point choices top-down
- Very compatible with Calling Context Trees



# Search Strategy: Hierarchical Search

- One of many choices
- Search trace point choices top-down
- Very compatible with Calling Context Trees





# Explaining Variation Using Key-Value Pairs in Trace Points

- Canonical Correlation Analysis (CCA)
- Used to find important key-value pairs in the traces

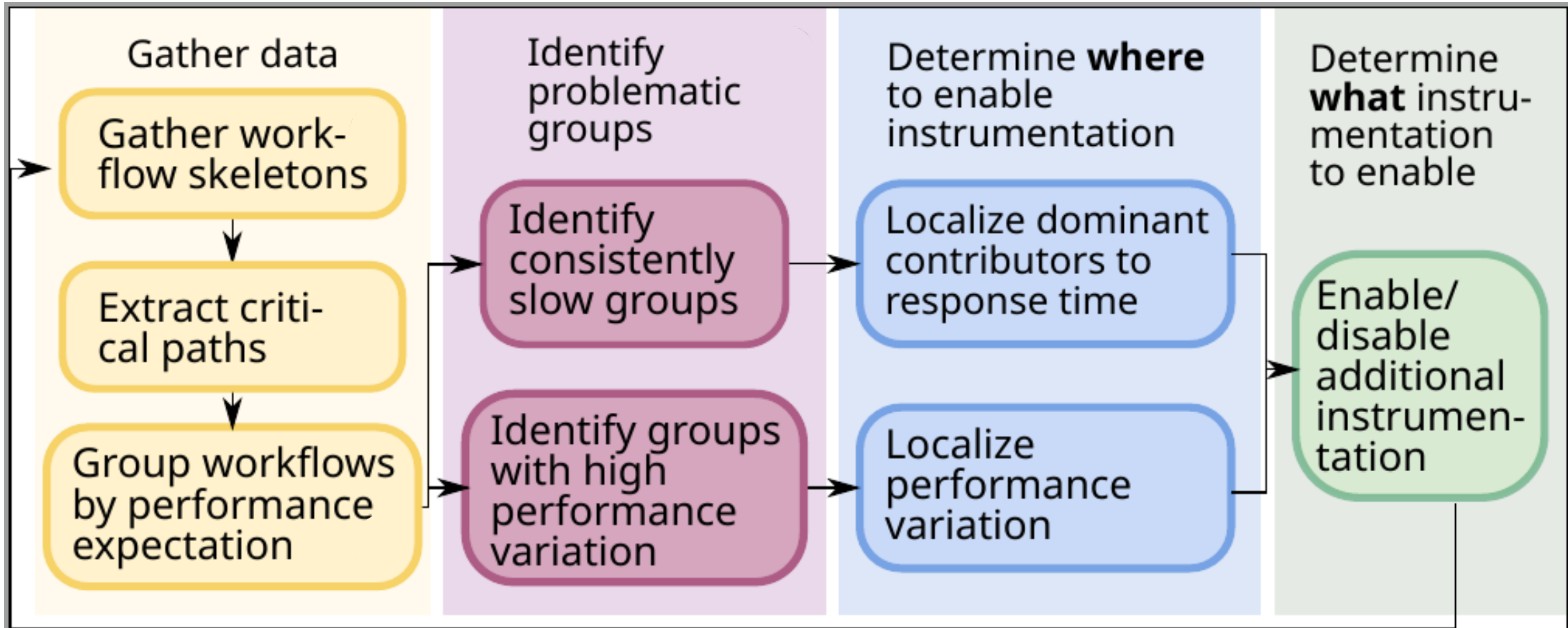
$$a' = \max_a \text{corr}(a^T X, Y)$$

$Y = (t_1, t_2, \dots, t_n)$  the request durations

$X = (x_1, x_2, \dots, x_m)$  the collected variables

$a' \in \mathbb{R}^m$  the coefficients indicating most correlated variables

## Vision of Pythia – Completing the Cycle



## Validating Pythia's Approach

- Can performance variation guide instrumentation choices?
- Run exploratory analysis for OpenStack
  - Start with default instrumentation
  - Localize performance variation
  - Find next instrumentation to enable
  - Use CCA for finding important key-value pairs

# Validating Pythia's Approach - Setup

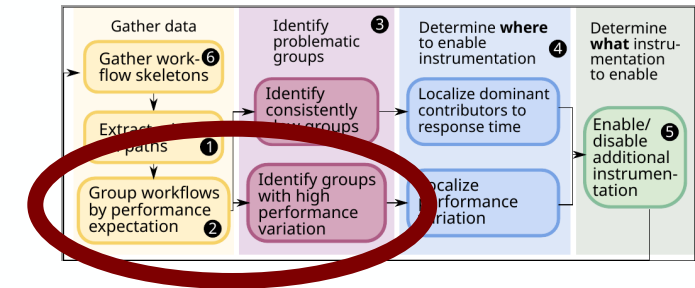
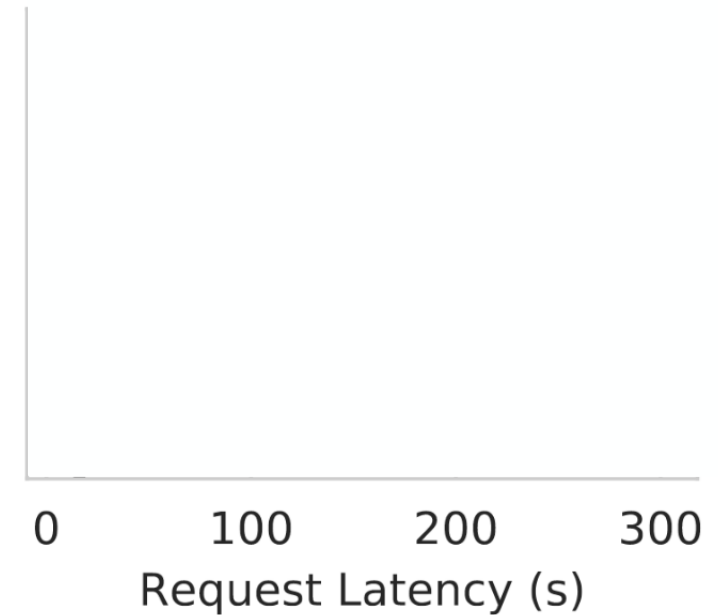
- **OpenStack:** an open source cloud platform, written in Python
- **OSProfiler:** OpenStack's tracing framework
  - We implemented controllable trace points
  - Store more variables such as queue lengths
- Running on MOC
  - 8 vCPUs, 32 GB memory
- Workload
  - 9 request types, VM/floating IP/volume create/list/delete
  - Simultaneously execute 20 workloads



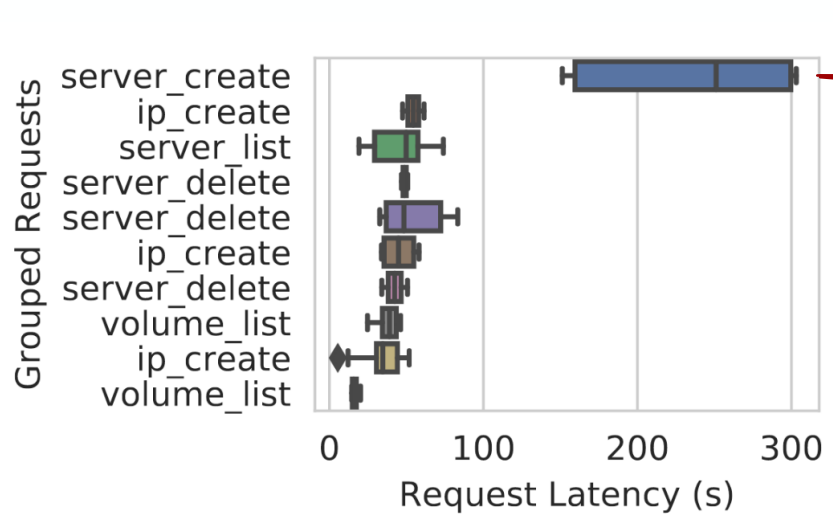
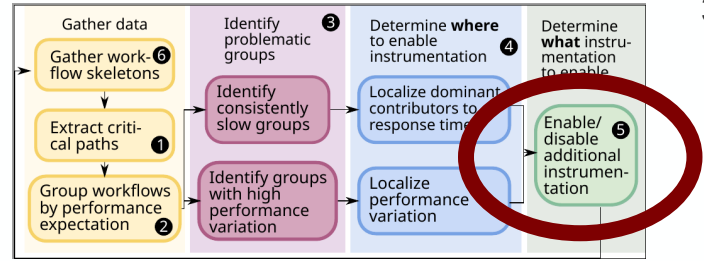
# Step 1: Grouping & Localization

- Collect latency values for each request
- Grouping: Same request type with same trace points
- Server create requests have unusually high variance and latency
- Pythia would focus on this group

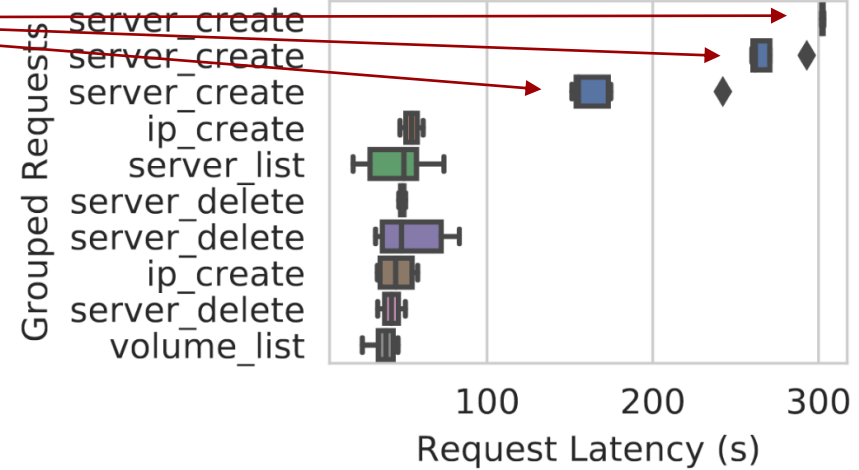
Grouped Requests



# Step 2: Enable additional instrumentation



Groups with different queue lengths



- Pythia localizes variation into a semaphore in server create
- After adding queue length variable into traces, we see 3 distinct latency groups
  - CCA also finds this variable important

**TAKEAWAY:** Pythia’s approach identifies the instrumentation needed to debug this problem

## Open Questions

- What is the ideal structure of the search space? What are possible search strategies? What are the trade-offs?
- How can we formulate and choose an “instrumentation budget”?
- How granular should the performance expectations be?
- How can we integrate multiple stack layers into Pythia?



## More in the paper

- Pythia architecture
- Problem scenarios
- Instrumentation plane requirements
- Cross-layer instrumentation

An automated, cross-layer instrumentation framework for diagnosing performance problems in distributed applications

Emre Ates<sup>1</sup>, Lily Suman<sup>2</sup>, Mert Toskani<sup>1</sup>, Oran Krieger<sup>3</sup>, Richard Magginnson<sup>4</sup>, Ayse K. Coskun<sup>1</sup>, Raja R. Sanyal<sup>5</sup>  
<sup>1</sup>Boston University, <sup>2</sup>Red Hat Inc., <sup>3</sup>Tufts University

**Introduction**

- Diagnosing performance problems in distributed applications is extremely challenging.
- It is hard to know where to place instrumentation a priori to help diagnose problems that may occur in the future.
- Python's automated instrumentation framework
- In response to a newly observed performance problem, it searches the space of possible instrumentation choices to enable the instrumentation needed to help diagnose it.

**Key enablers**

- Workflow-centric tracing (i.e., end-to-end tracing)
- High performance, variation among requests that are expected to perform similarly

**Design**

Components in the instrumentation plane are provided by different workload-centric tracing infrastructures

**INSTRUMENTATION PLANE**

- Traces in the form of DAGs
- Concurrency/parallelization
- Hierarchical relationships
- Cache Coherence
- Dependencies, flow control
- Control plane

**CONTROL PLANE**

- Enable trace points to localize problems
- Subsets enabled instrumentation to pre- and post- (e.g., JK, VMS)
- Apply search strategies

**Vision**

- Python will **selectively enable and disable** trace points added to distributed applications and lower stack layers
  - Trace points can consist of variable values, e.g., function parameters, queue lengths, performance counters.
- Python will take as input
  - Initial, low-fidelity expectations
  - Workflow sessions
- Initial expectations are specified by developers based on properties of request/critical paths
  - E.g., Expect same request types to perform similarly

**How could Python aid?**

**Problem in OpenStack: "SERVER CREATE" requests show high variance!**

- Extract critical paths
- Group traces by their type (e.g., SERVER CREATE, SERVER DELETE)
- Identify problematic group -> SERVER CREATE
- Localize the variation to the machines involved in the flow service
- Enable trace points within the function points to a sampler

Show that the queue length variable highly correlated with request latency

**Search Space & Strategies**

**Search Space:**

- Enumerated: Crawl all trace points that are direct children of the problem edges in the DAG.
- Binary search: Hierarchical search can be accelerated by skipping layers when exploring.
- Cross-layer: Instrumentation on different stack levels may be prioritized after the problem has been localized to a single VM.
- Covariance: For high variance groups, the covariances of edge pairs will be calculated.

**Search Strategies:**

- Enumerated: Crawl all trace points that are direct children of the problem edges in the DAG.
- Binary search: Hierarchical search can be accelerated by skipping layers when exploring.
- Cross-layer: Instrumentation on different stack levels may be prioritized after the problem has been localized to a single VM.
- Covariance: For high variance groups, the covariances of edge pairs will be calculated.

**Results**

- Python's approach for debugging the contention problem
  - High variance in SERVER CREATE group
  - 98% of the variance is within 6 edges (SERVER CREATE)
  - The edge with highest variance corresponds to semaphore
  - CCA shows that **queue length** variable correlates the most (CIS with P value 1e-1)

**Conclusion**

- It is challenging to decide where to instrument upfront
- We presented initial steps toward creating an **automated instrumentation framework that explores the search space automatically**

**OPEN QUESTIONS:**

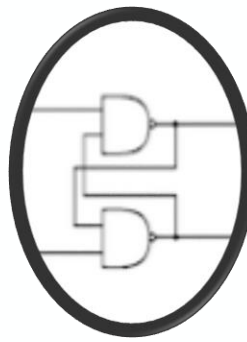
- How detailed do initial expectations need to be for Python?
- Out of the many possible search-space representations and search strategies, which ones are most useful?



Red Hat



- It is very difficult to debug distributed systems
- Automating instrumentation choice is a promising solution to overcome this difficulty



# Concluding Remarks

More info in our paper ([bu.edu/peaclab/publications](http://bu.edu/peaclab/publications))

Please send feedback to [ates@bu.edu](mailto:ates@bu.edu) or join us at the poster session