Debugging *Complex* Systems

Terran Melconian
Velocity New York
October 4, 2017

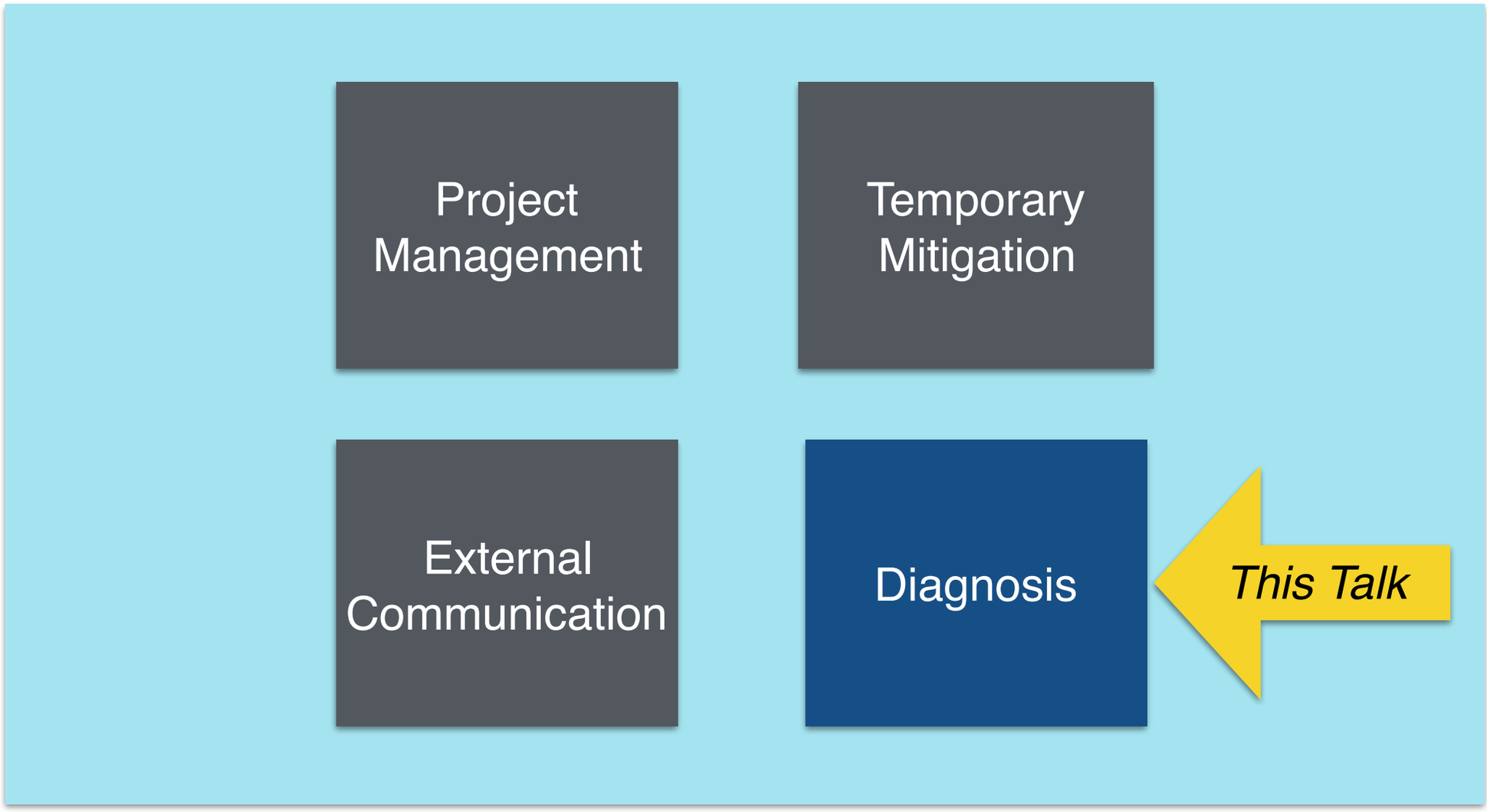
terran@airnetsim.com twitter @terranmelconian

- Aeronautics
- Electronics
- Development
- Operations
- Data science
- Grey beard

What's Complex?

- Multiple interacting components
- Emergent behavior
    - Dynamics occur which were not intentionally designed in
- Usually larger than a single piece of software on a single host

Incident Response



Applicability

- To any system which once worked or sometimes works and then does not
- Which has been or can be observed and measured in both states

Core Principles

- Question beliefs
  - *You believed* the system worked. It doesn't.
  - Your other beliefs are not magically better.
- Divide the problem space
  - Binary search beats linear search
- Fast measurements first
  - Look up a measurement we have: 5 min
  - Write, review, deploy, wait for peak: 24+ hours

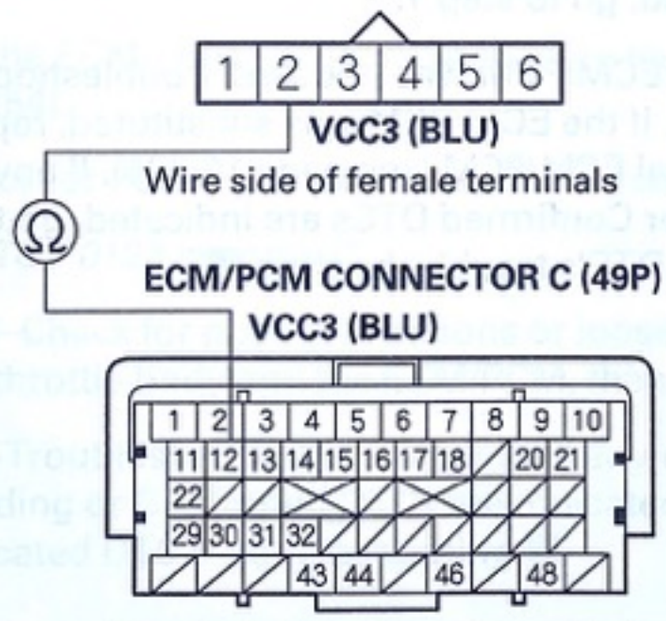
Teaching the Skill

- Uncorrelated with software development
- Contrast actual states - avoid distracting contrasts with idealized “shoulds”
  - smallest set of changes to reproduce
- Enforce shared, written records and diagrams
- Practice in advance of critical failures

Domains

- Computer systems
- Your car
- Health
- Leaks in your house
- Almost any kind of system which *used to work*

15. Check for continuity between ECM/PCM connector terminal C12 and throttle body 6P connector terminal No. 2.



Is there continuity?

YES–Go to step 23.

NO–Repair open in the wire between the throttle body and the ECM/PCM (C12), then go to step 18.

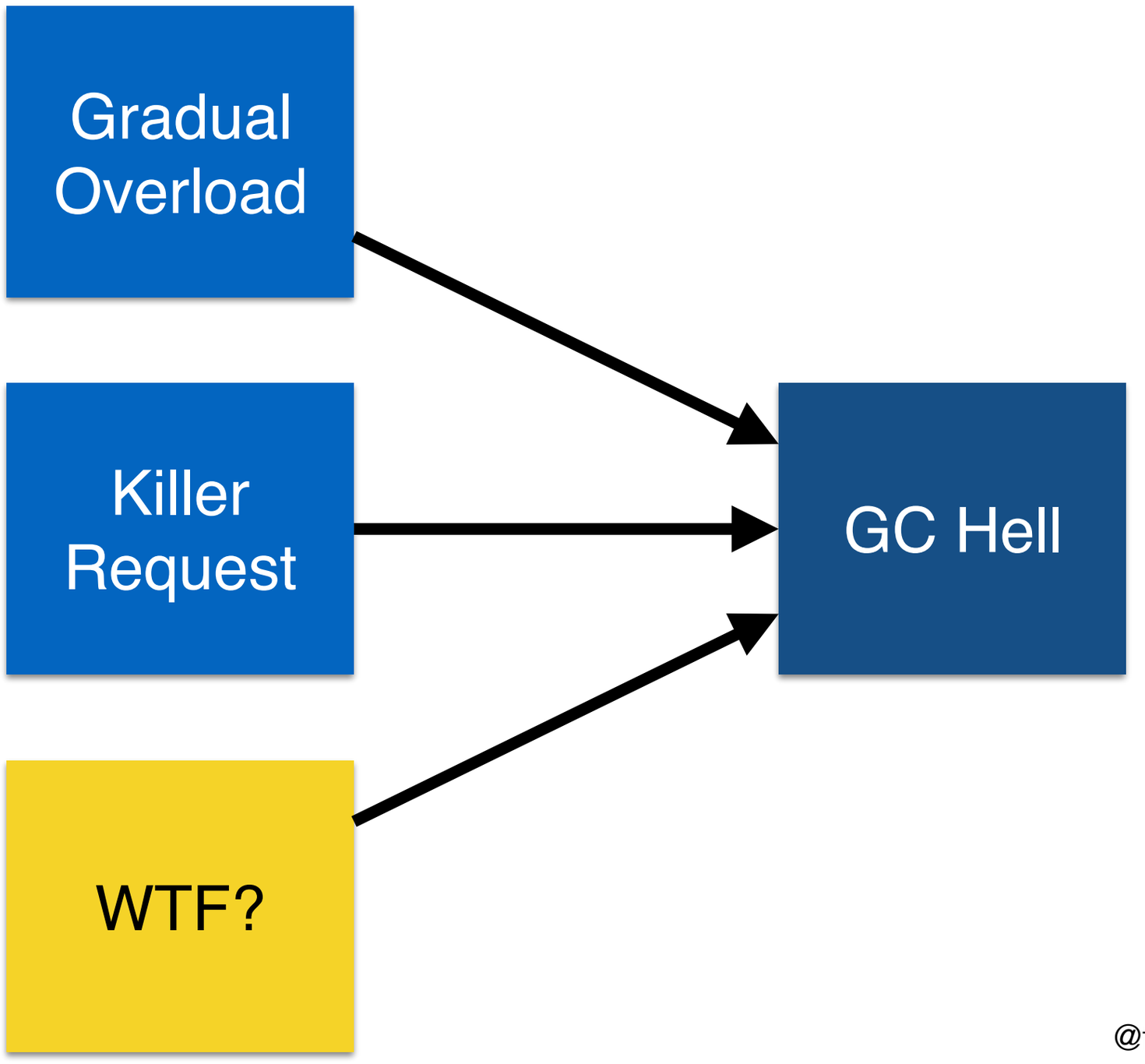
Start with Symptom

- Errors
- Slow Response Time
- Server crash
- GC hell
- Bad data

Mechanism

- Draw a tree of possible causality
  - Rooted at symptom
  - Possible causes point to root
  - Causes of those causes and so on
- Take organized data samples
  - Add columns as you expand your analysis

Example: GC Hell

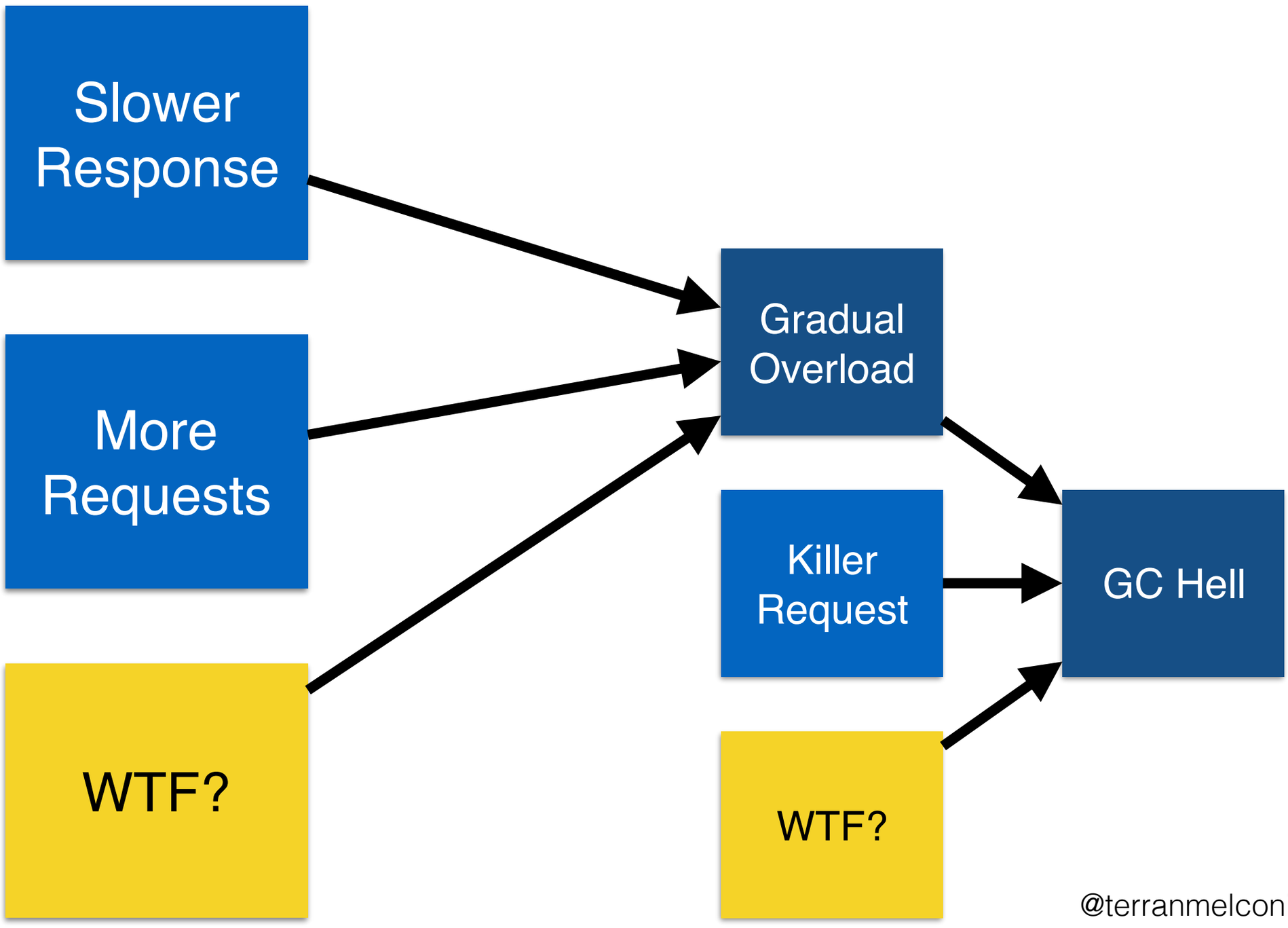


GC Hell Data

Time	GC CPU %
13:46 (failure)	100%
13:40	3%
13:00	4%
12:00	3%

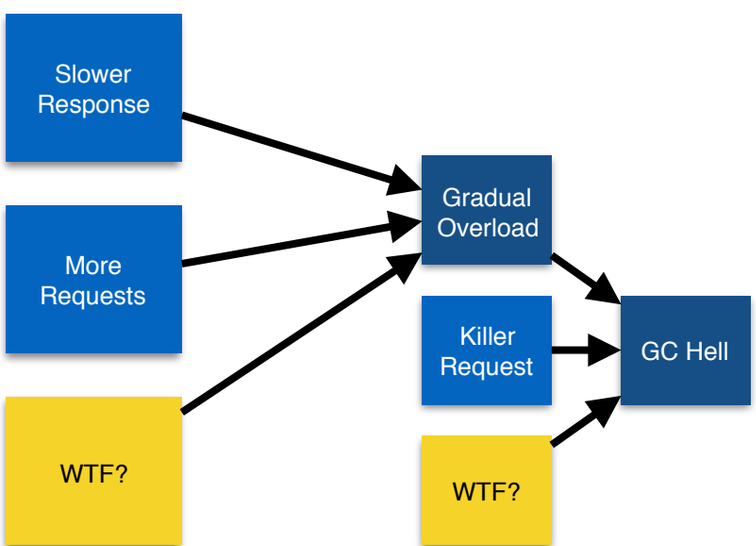
Time	GC CPU %
13:46 (failure)	100%
13:40	92%
13:00	34%
12:00	3%

GC Hell Step 2

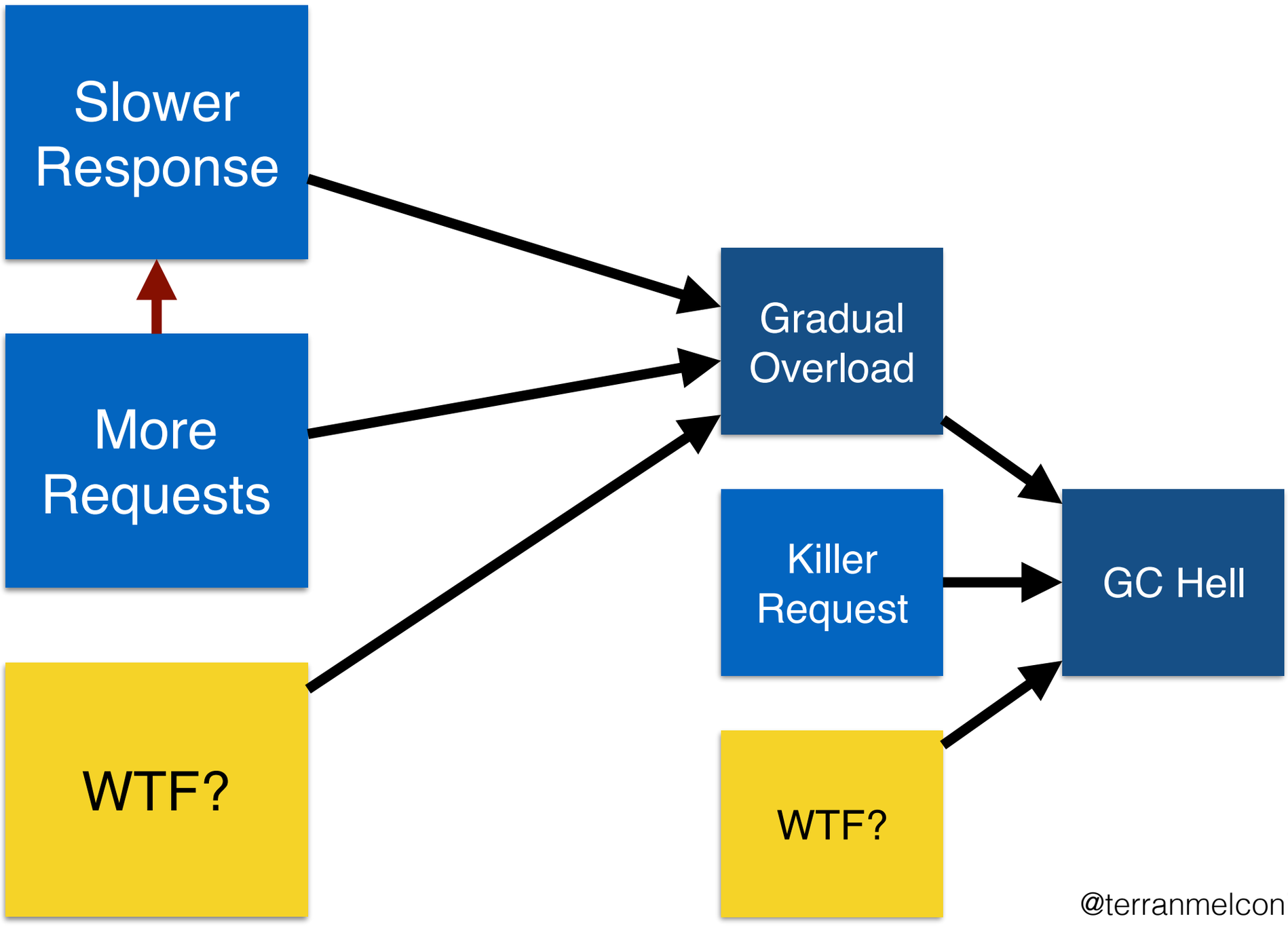


Step 2 Data

Time	GC CPU %	Response Time
13:46 (failure)	100%	16423 ms
13:40	92%	2473 ms
13:00	34%	844 ms
12:00	3%	192 ms



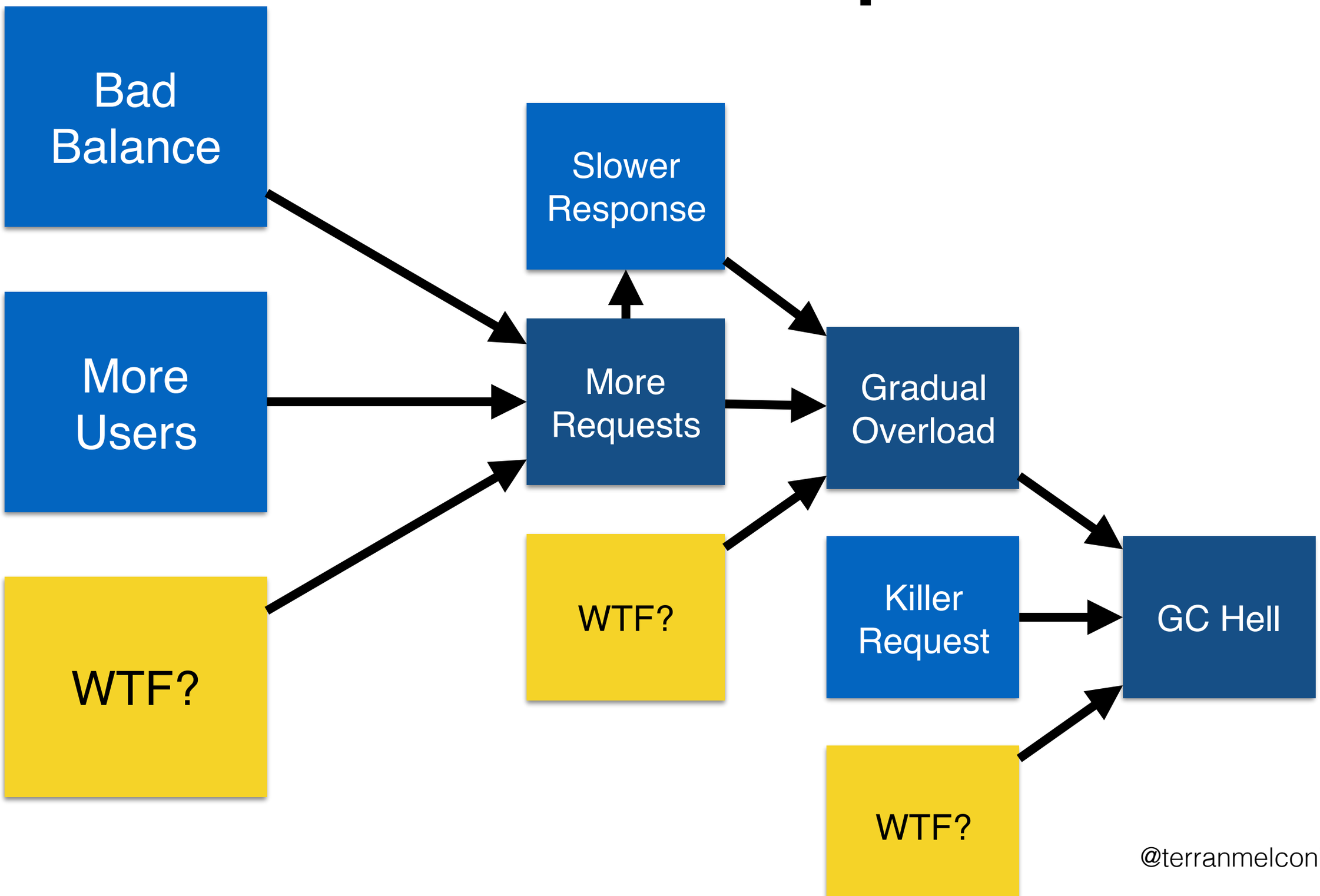
GC Hell Step 2 bis



Step 2 Data bis

Time	GC CPU %	Response Time	Req/Min
13:46 (failure)	100%	16423 ms	3
13:40	92%	2473 ms	352
13:00	34%	844 ms	1630
12:00	3%	192 ms	850

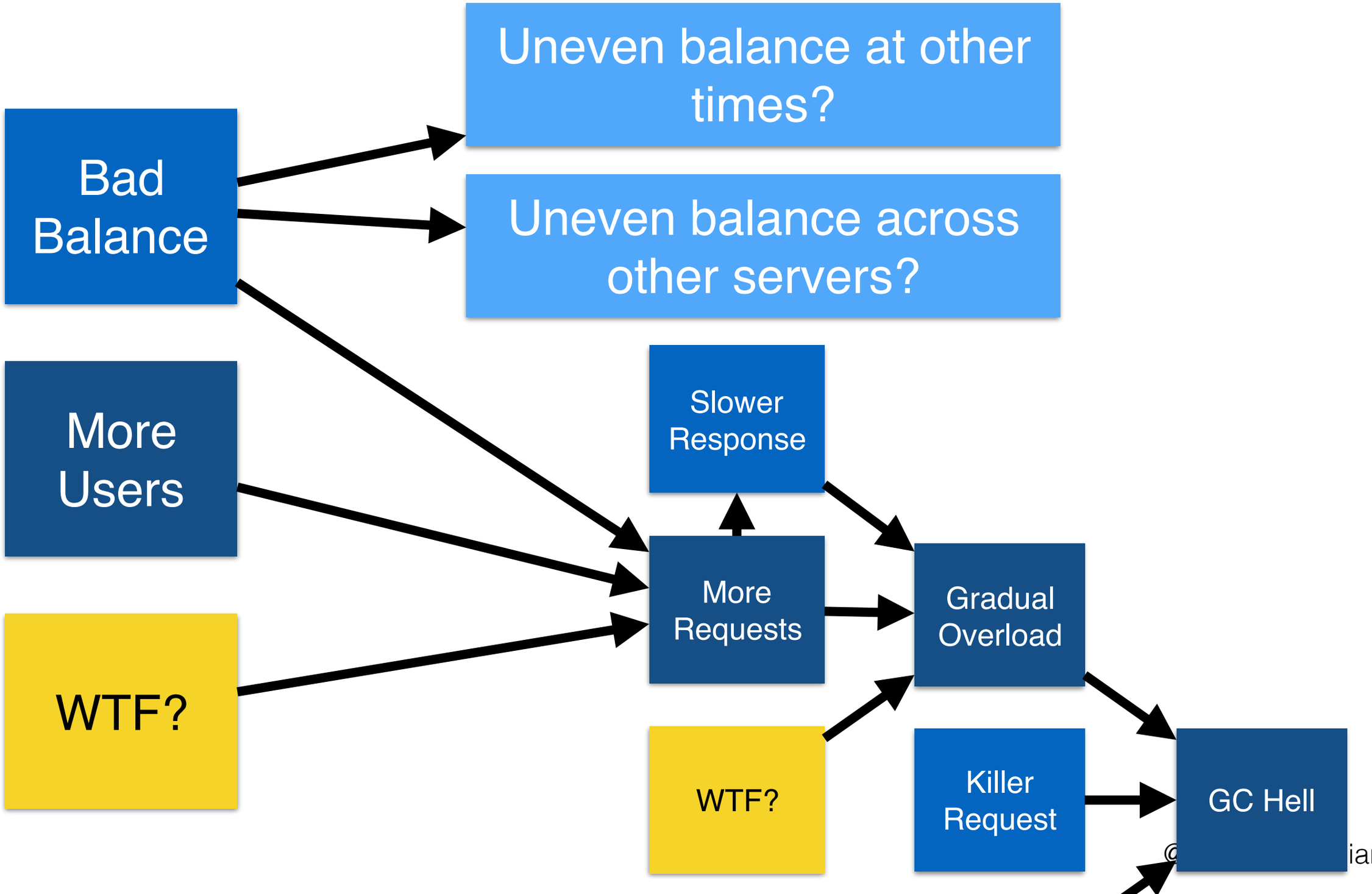
GC Hell Step 3



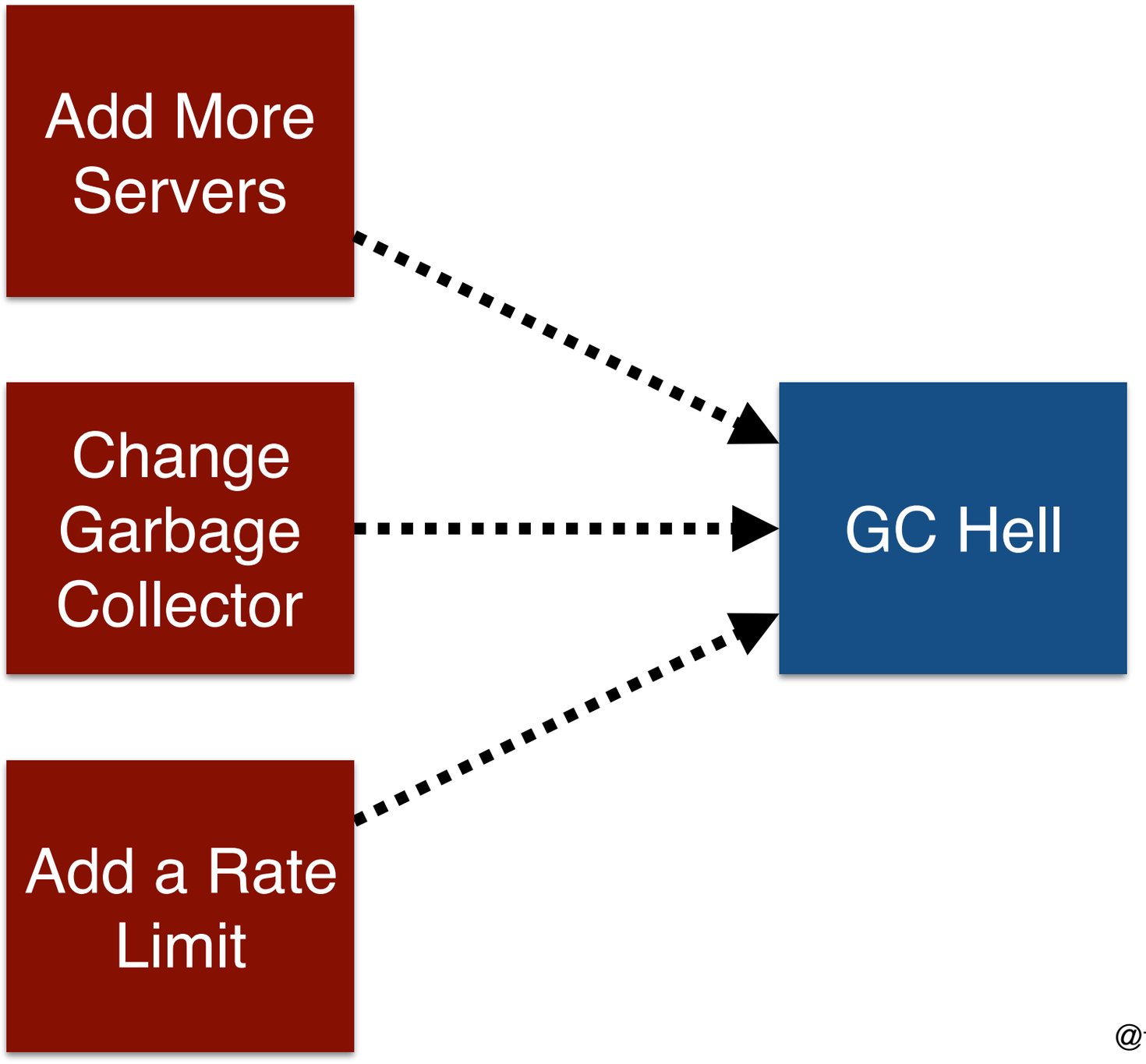
Step 3 Data

Time	Server A			Server B		
	GC CPU %	Response Time	Req/Min	GC CPU %	Response Time	Req/Min
13:46 (failure)	100%	16423 ms	3	15%	302 ms	1250
13:40	92%	2473 ms	352	3%	240 ms	1002
13:00	34%	844 ms	1630	4%	180 ms	702
12:00	3%	192 ms	850	3%	201 ms	842

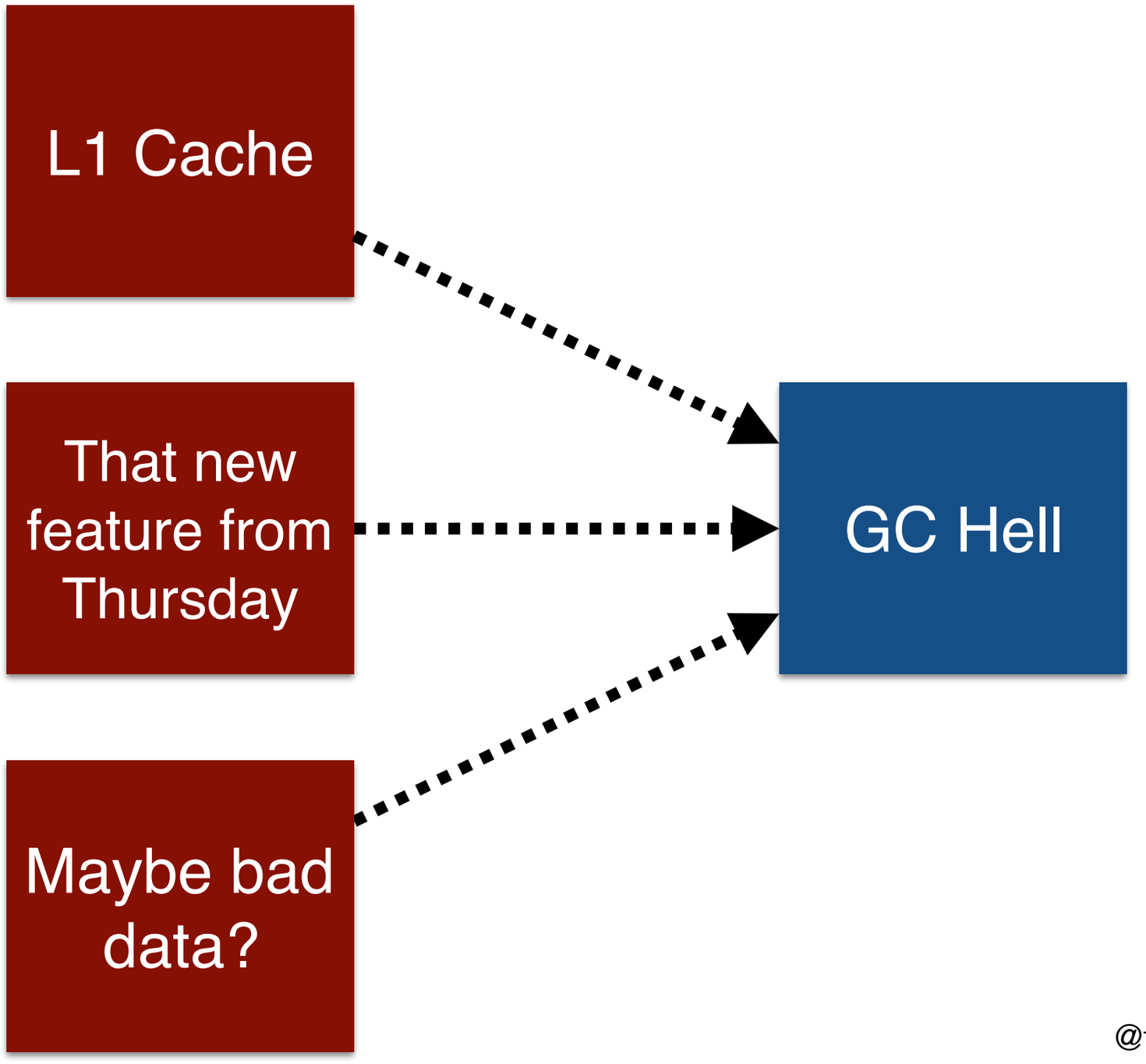
GC Hell Step 4



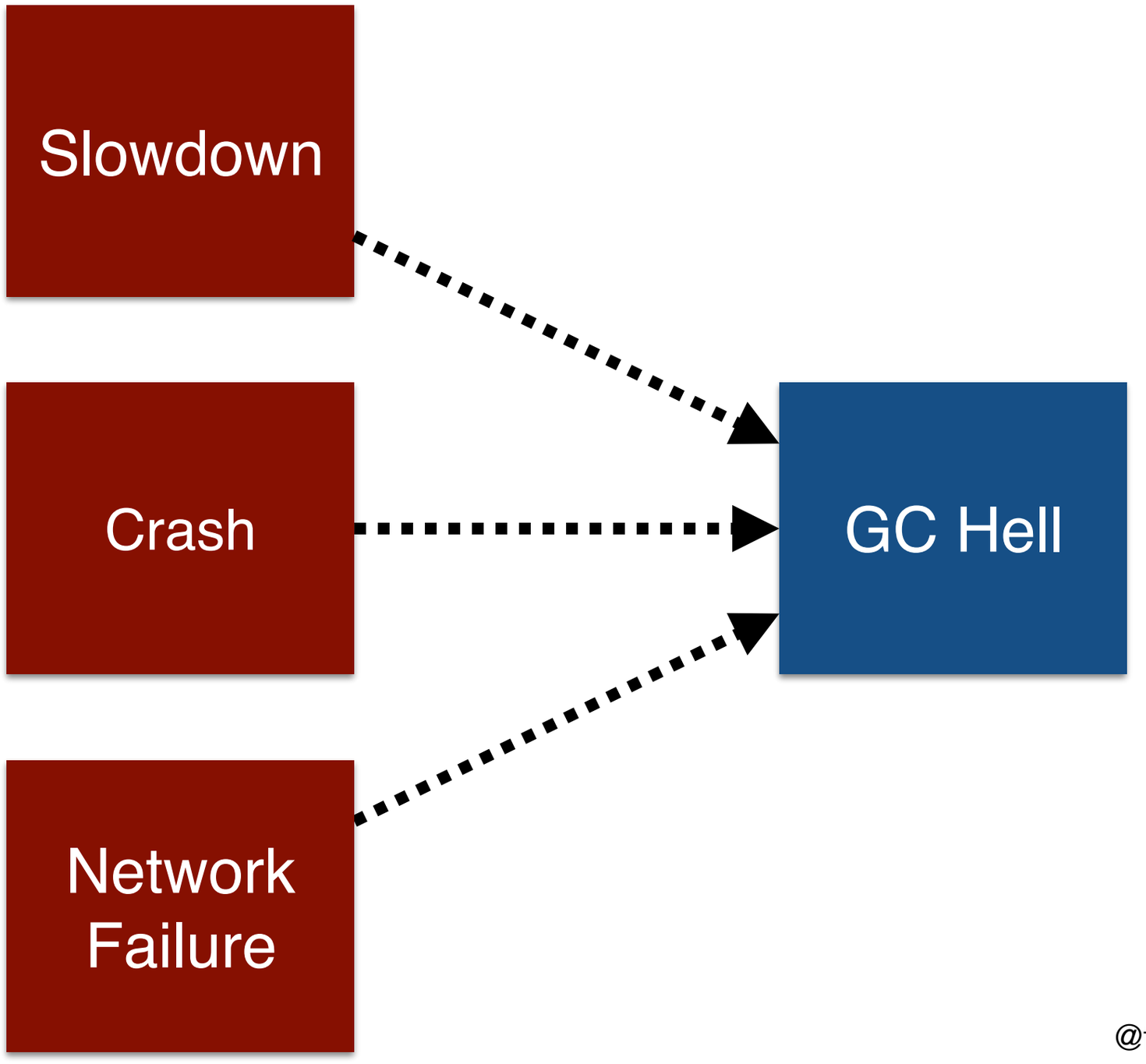
Anti-Example 1



Anti-Example 2



Anti-Example 3



Partitioning

- Do
  - By system component
  - By service
  - By time of checkin/deployment
    - beware, blind to causes which are not a code change
- Don’t
  - By ways to mitigate
  - By listing individual pieces of code
  - By ignoring the information in the symptoms

Extensions

- Weight the tree by prior beliefs and partition weight instead of node count
  - Default first steps such as rolling back release
    - but limit your temptation to repeatedly pursue high-confidence guesses
    - disagreements over weight more likely
- Give tree and process to others for diagnosis
- Plan your logging and dashboards

Summary

- Start with a symptom
- Draw a tree of possible causes
- Take measurements to partition the tree
- Prefer observing to mutating and waiting
- Record all your data in one place
- Suspect everything you believe

Questions?

terran@airnetsim.com twitter @terranmelconian

slides: from O’Reilly site or http://www.airnetsim.com/terran/