Analysis and Design of Scalable Programming Auto grader

MTP Final Thesis Presentation

Guide:
Prof Varsha Apte

Submitted by:
Chaitanya Varma
203050026

Table of Contents

01

Need for scalability in Programming auto-graders

02

Evalpro background

03

Prior work of Evalpro Scalability

04

MTP Phase-1

05

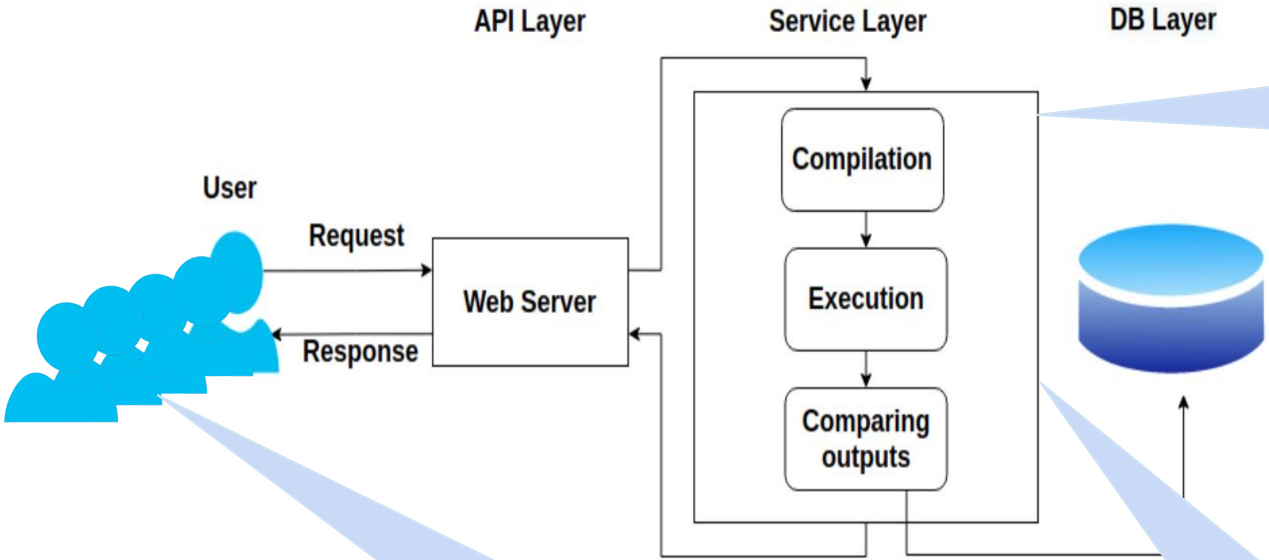
MTP Phase -2

06

Conclusion & Future Work

Need for Scalability in Programming auto-graders

Typical programming auto grader architecture



processing time depends upon the user code and cannot be profiled

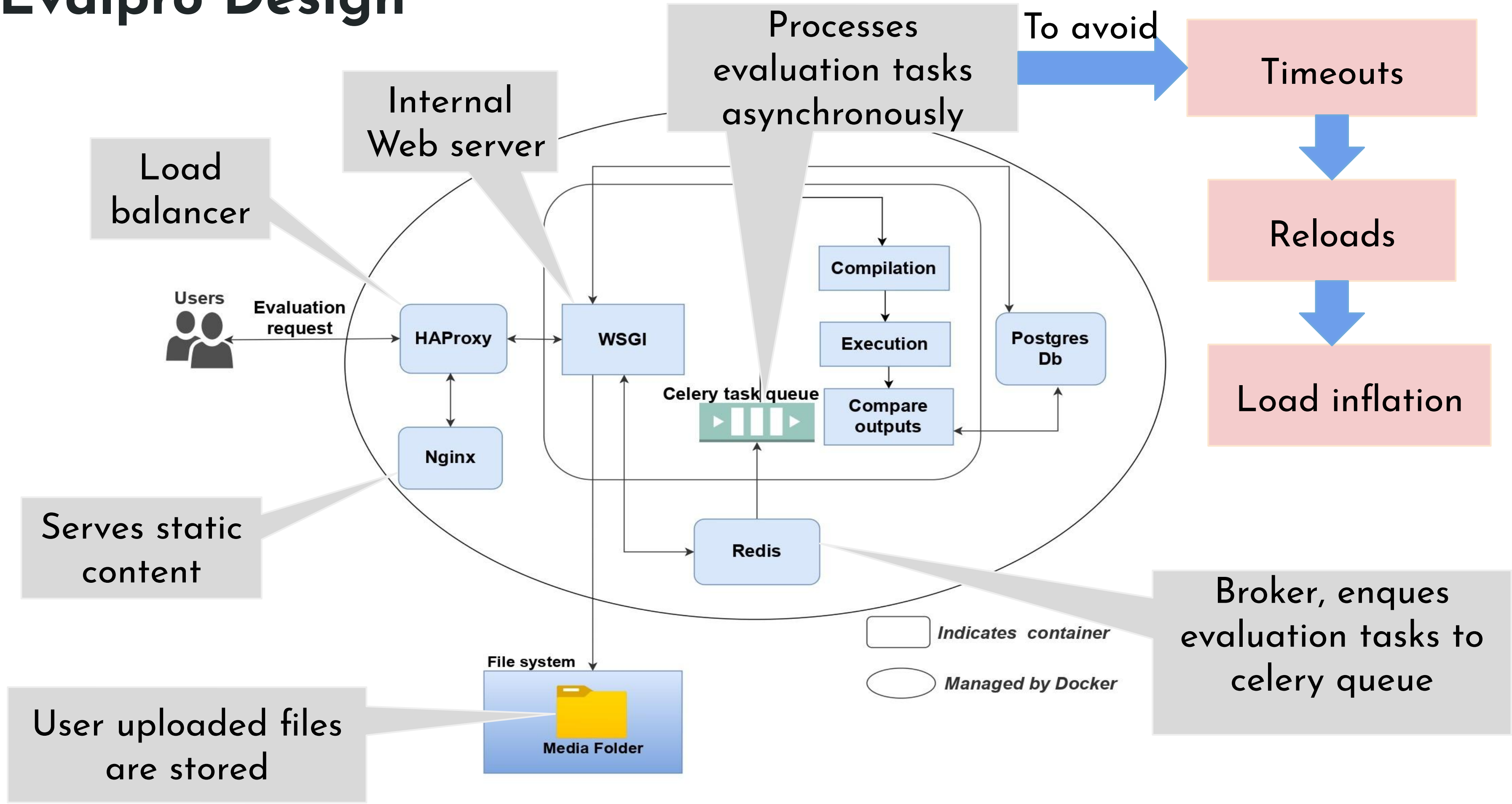
Scalability: Ability of the system to scale its performance with increase in resources

At peak usage, system need to scale its performance

Not possible to estimate resource requirement in prior

Background about Evalpro

Evalpro Design



Load balancer

Internal Web server

Processes evaluation tasks asynchronously

To avoid

Timeouts

Reloads

Load inflation

Serves static content

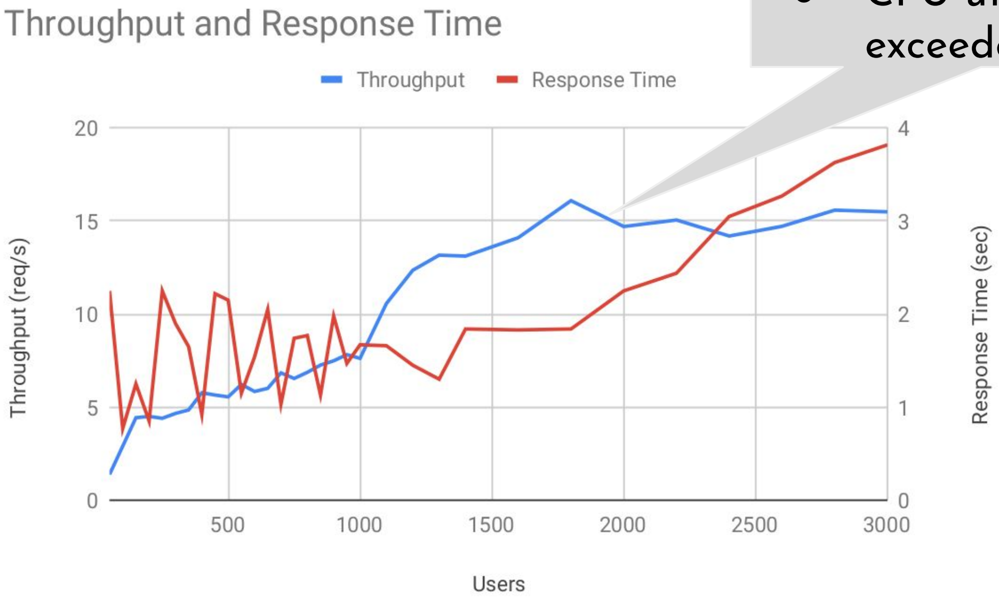
User uploaded files are stored

Broker, enques evaluation tasks to celery queue

Prior work of Evalpro scalability

Prior work (MTP 2019) and its limitations

40 CPU core server



- Throughput flattened
- CPU utilization not exceeded 5%

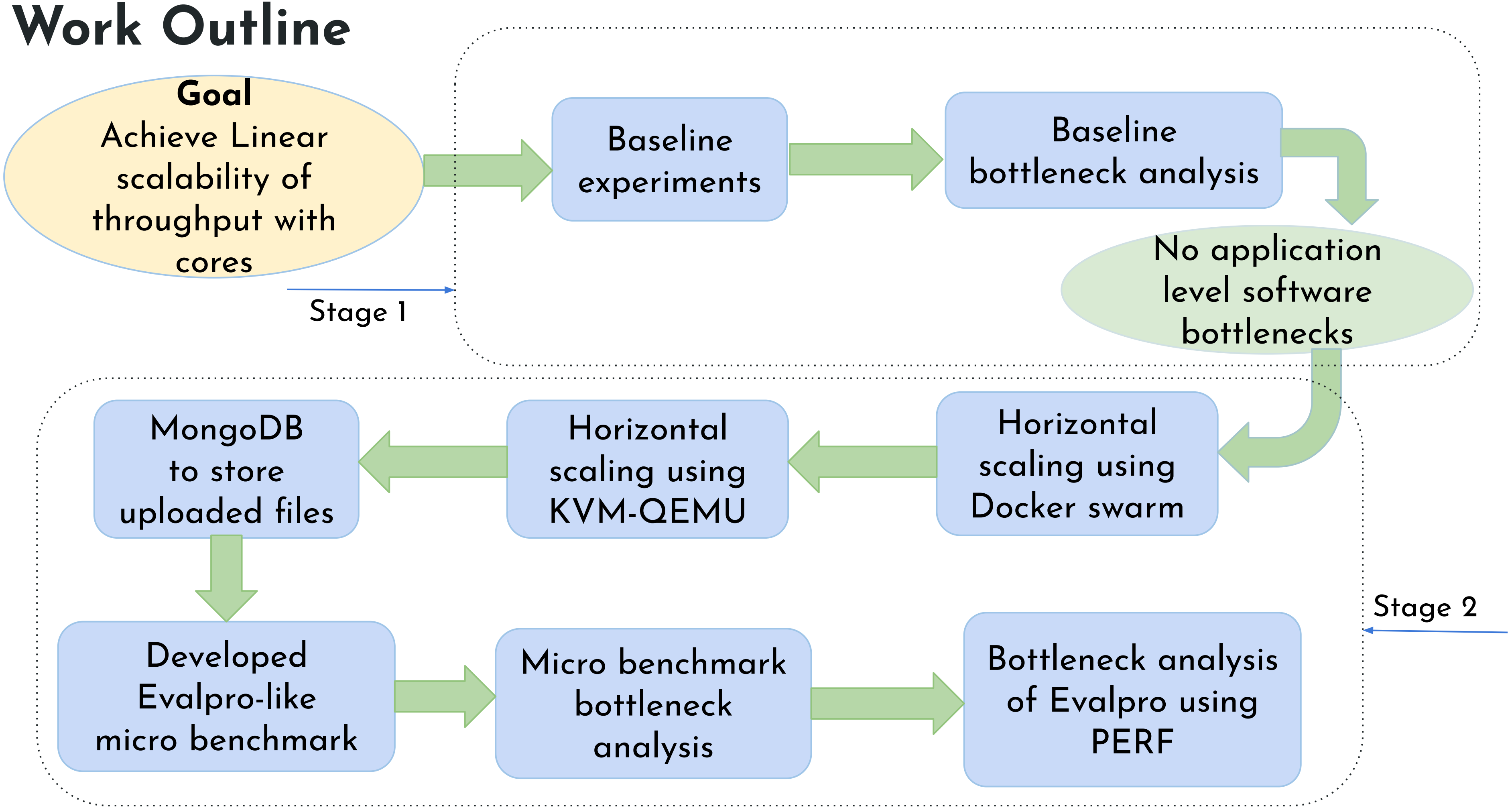
Scalability issue

Increasing WSGI+Celery replicas increased the throughput and CPU utilization

- Reason for improvement in scalability not justified
- Scalability issue with single replica, configurations affecting the performance not studied

“The data and figures in this slide were taken from Anshul MTP thesis 2019”

Work Outline



Goal
Achieve Linear scalability of throughput with cores

Stage 1

Baseline experiments

Baseline bottleneck analysis

No application level software bottlenecks

Stage 2

Horizontal scaling using Docker swarm

Horizontal scaling using KVM-QEMU

MongoDB to store uploaded files

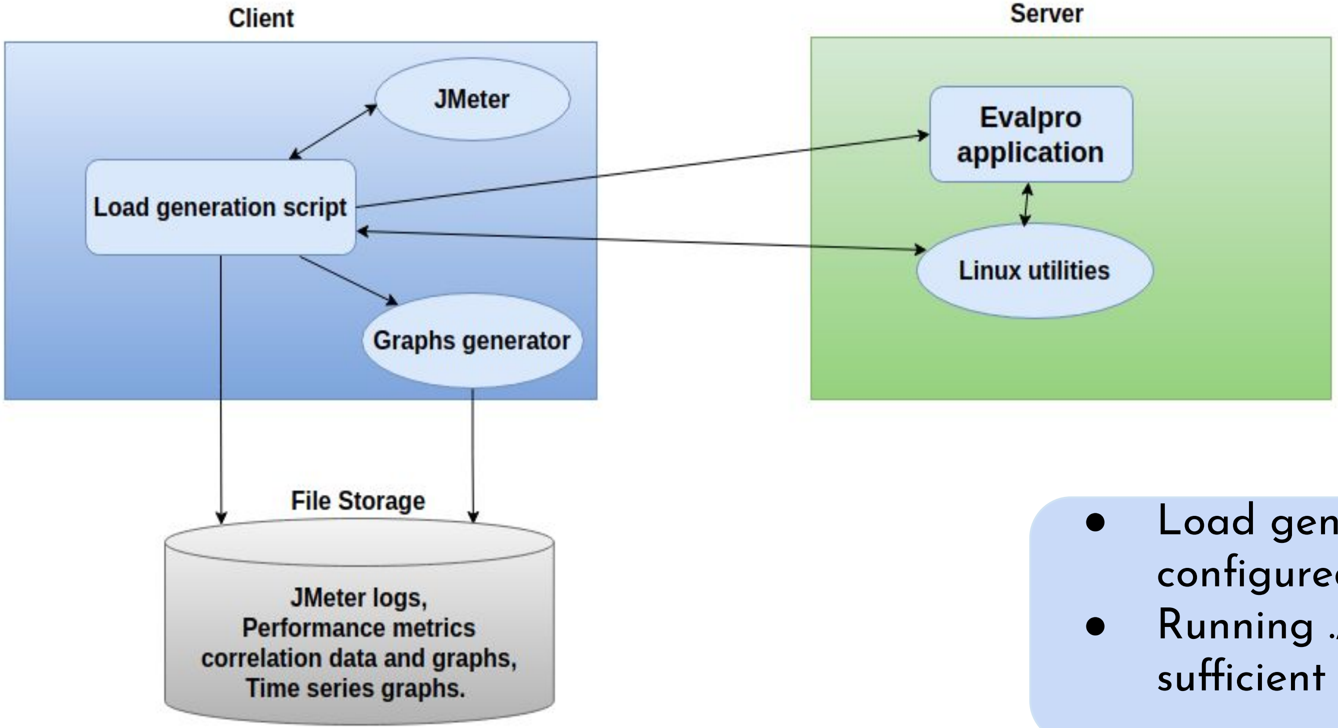
Developed Evalpro-like micro benchmark

Micro benchmark bottleneck analysis

Bottleneck analysis of Evalpro using PERF

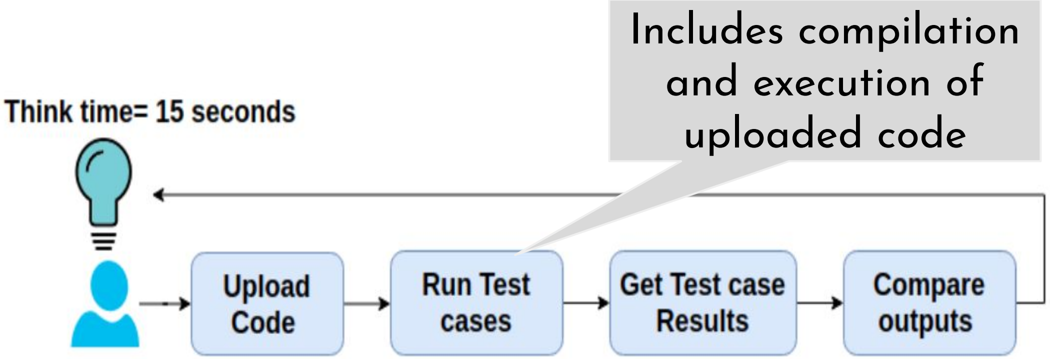
MTP stage 1

Load generation and performance measurement Infrastructure

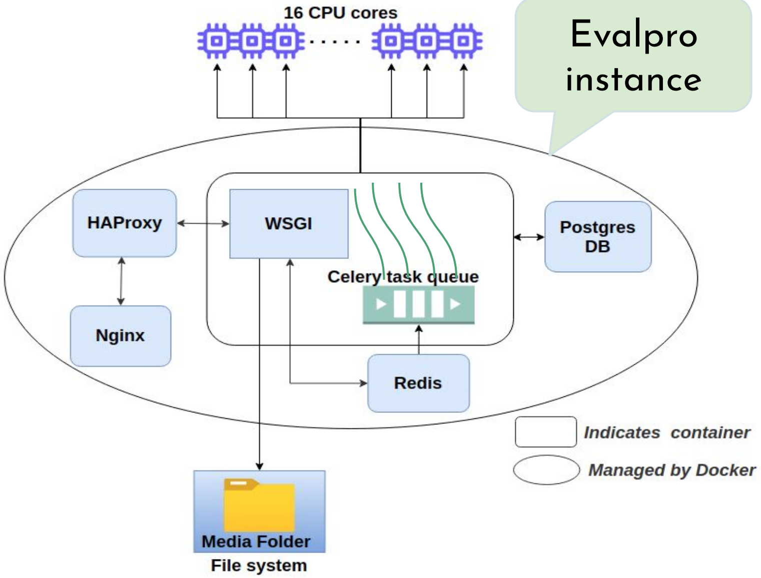


- Load generation configured for 5 minutes
- Running ./load_test.sh is sufficient

Baseline Experiment setup


Includes compilation and execution of uploaded code

Realistic user session


Evalpro instance

Type	CPU	Cores	Memory	L3 cache	L2 cache	L1 cache
Server	Intel^{R} Xeon^{R} CPU E5-2650 v2 @ 2.60GHz	16	16GB	20MB	256KB	32KB
Client	AMD Opteron^{TM} Processor 6212	16	16GB	6MB	2MB	64KB

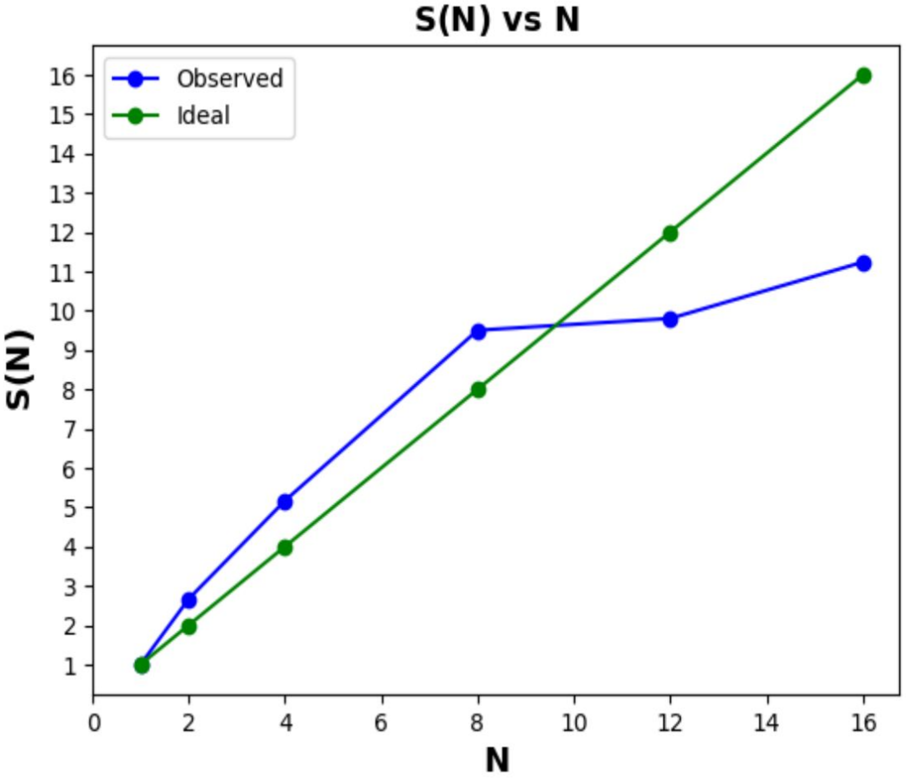
Hardware specifications for baseline experiments

Baseline 16

To avoid obvious bottlenecks,
Postgres DB connections set to 10000
Celery threads set to number of cores

Throughput$_{max}$(N) : Maximum throughput with N CPU cores
S(N) : Throughput scalability factor with N CPU cores

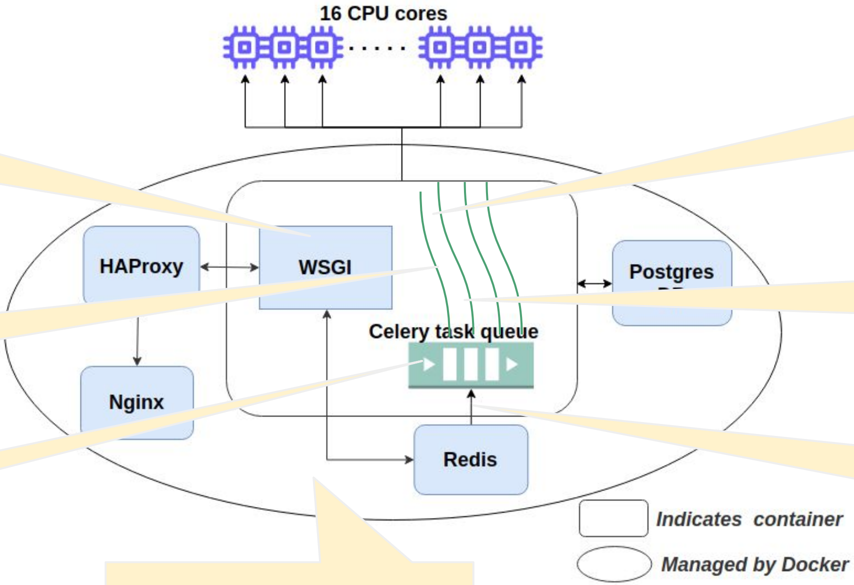
$S(N) = Throughput_{max}(N) \div Throughput_{max}(1)$
Ideal $S(N) = N$



Throughput$_{max}$(1)	*Throughput$_{max}$(16)*	Ideal *Throughput$_{max}$(16)*	*S(16)*
0.3 req/sec	3.37 req/sec	4.8 req/sec	11.23

Linear scaling of throughput not achieved

Baseline bottleneck analysis



Increased gunicorn workers

Increased prefetch count

Made celery queue transient

Disabled logging

Setting celery CPU affinities

Increased celery threads

Increased open connections to redis

None of these experiments improved throughput scalability

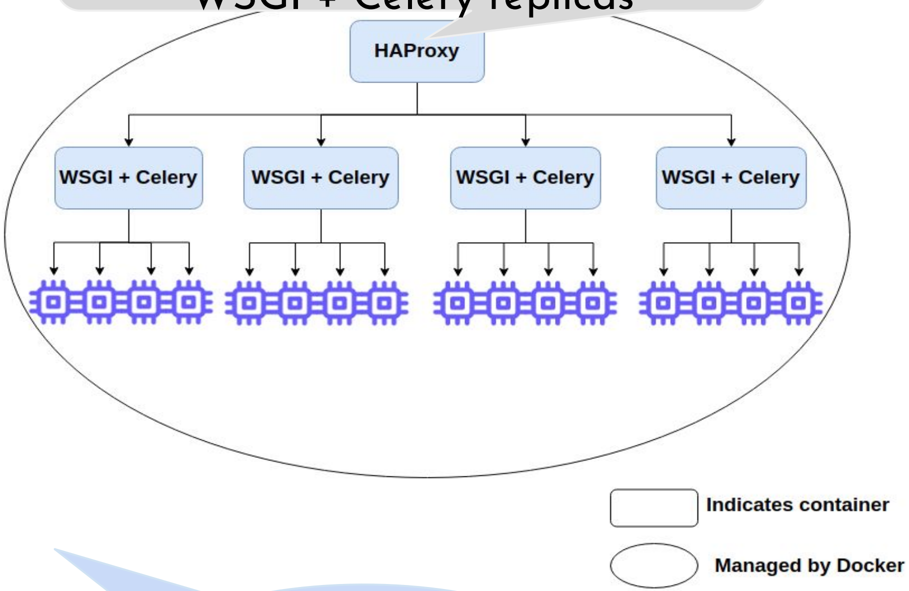
No application level software bottlenecks

All cores 90-100 % utilized

MTP Stage-2

Horizontal scaling with Docker swarm

Load balances request load across multiple WSGI + Celery replicas

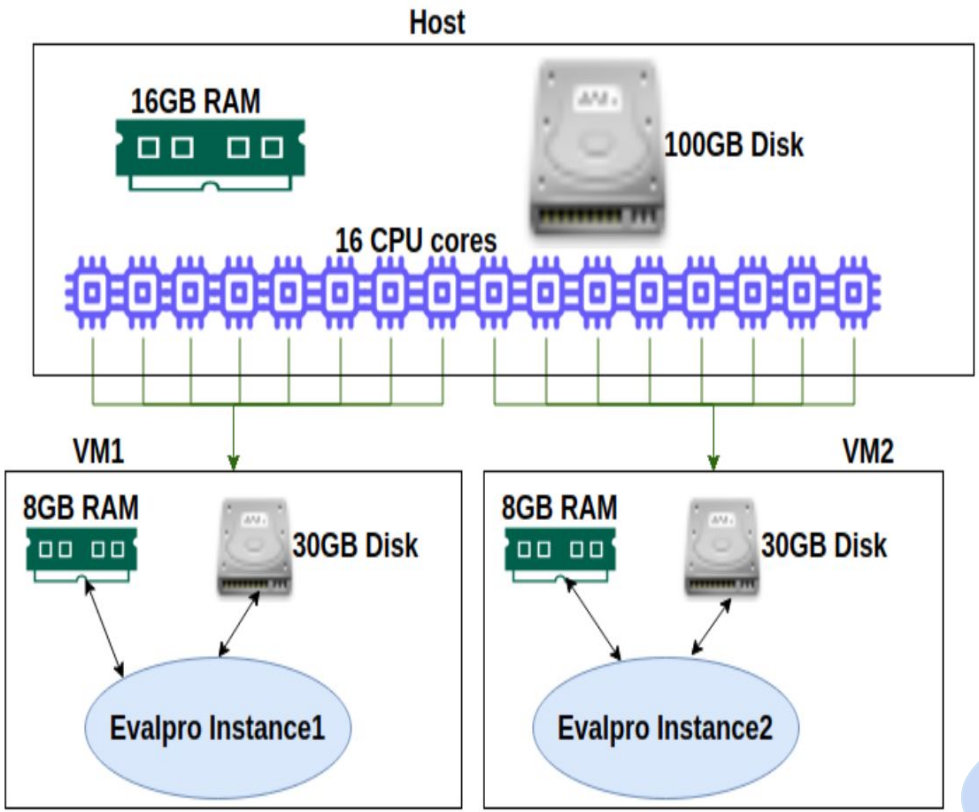


	Baseline	Using Docker swarm
Throughput$_{max}$(1) (req/sec)	0.3	0.3
Throughput$_{max}$(16) (req/sec)	3.37	3.4
Ideal *Throughput$_{max}$(16)* (req/sec)	4.8	4.8
S(16)	11.23	11.3

Docker swarm didn't improve throughput scalability

Horizontal scaling with KVM-QEMU

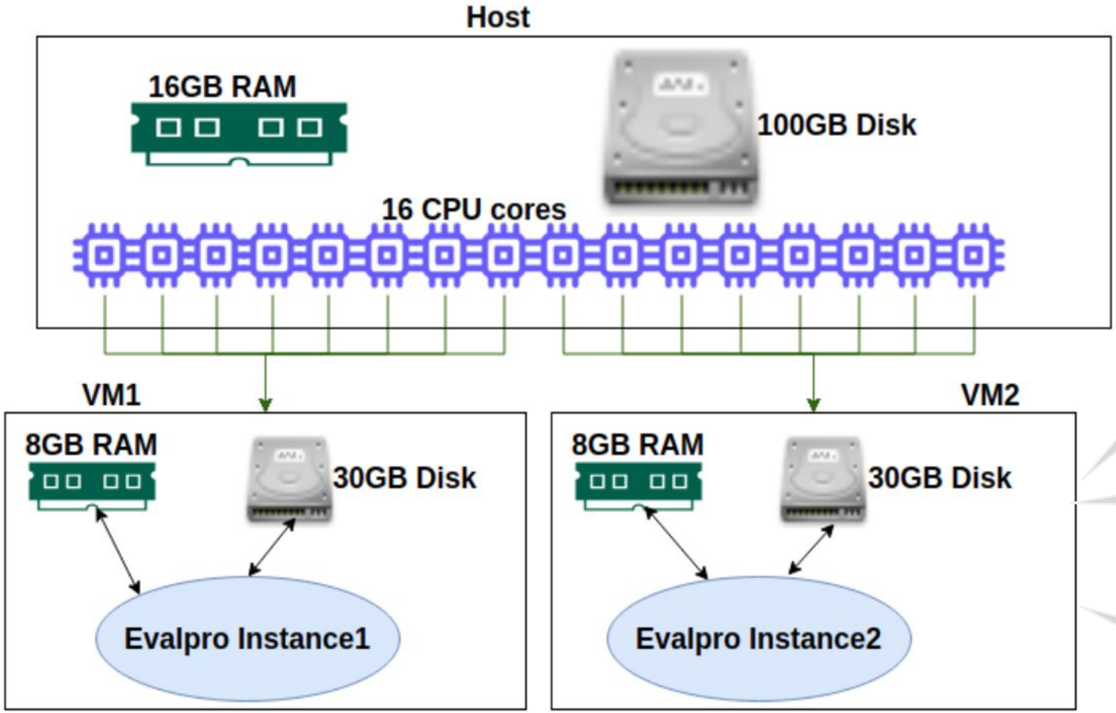
Completely Isolated VM setup



	Baseline	Isolated VM setup
Throughput$_{max}$*(1)* (req/sec)	0.3	0.3
Throughput$_{max}$*(16)* (req/sec)	3.37	4.45
Ideal *Throughput*$_{max}$*(16)* (req/sec)	4.8	4.8
S(16)	11.23	14.5

Isolated setup improved throughput scalability

Completely Isolated VM setup limitations



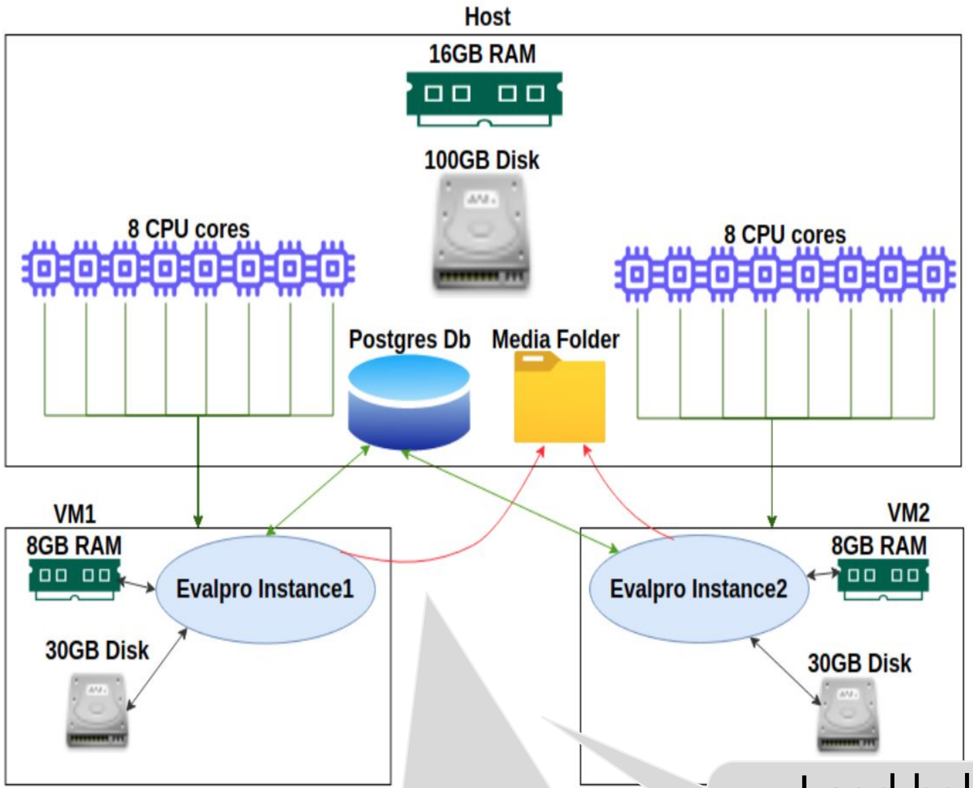
Each VM need to maintain separate user related data

Load balancer need have user to VM mapping

Migrating user data with new VM addition is very complex and not feasible.

Not the desired solution

User data and files sharing VM setup



	Baseline	Data and file sharing VM setup
$Throughput_{max}(1)$ (req/sec)	0.3	0.3
$Throughput_{max}(16)$ (req/sec)	3.37	2.78
Ideal $Throughput_{max}(16)$ (req/sec)	4.8	4.8
$S(16)$	11.23	9.62

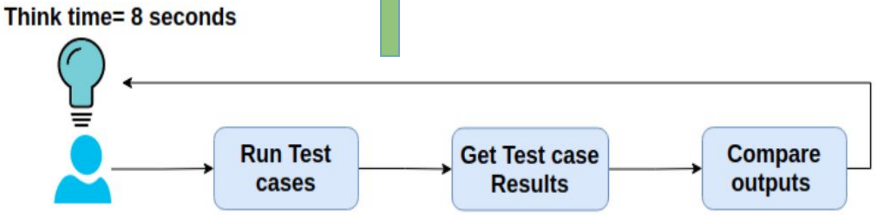
Adding new VM doesn't require migration of user data

Load balancer need not maintain user to VM mapping

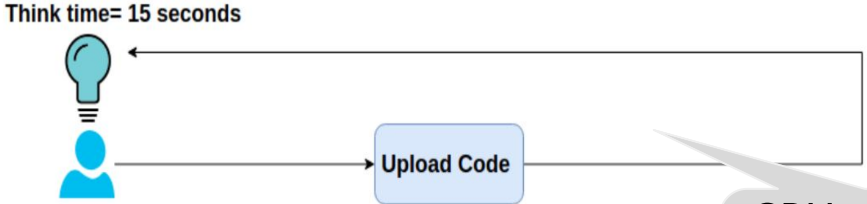
Throughput scalability decreased with this setup

Reason for reduction in throughput scalability

Performed experiments with different user sessions

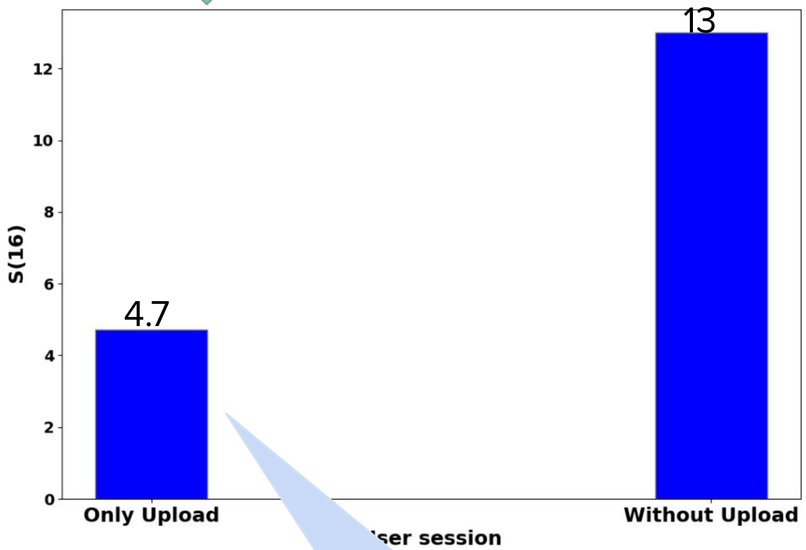


User session: Without Upload



User session: Only Upload

CPU utilization never exceeded 35%



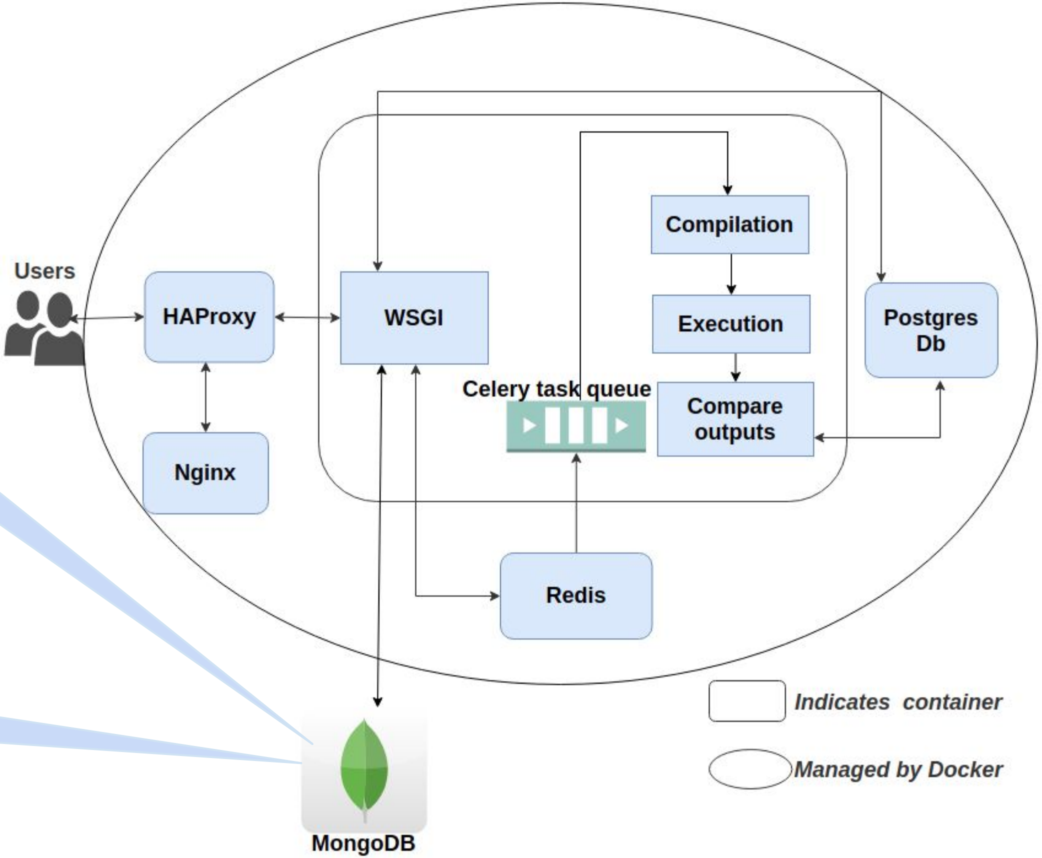
Uploading file by VMs in host is reducing scalability

Using MongoDB for File Storage

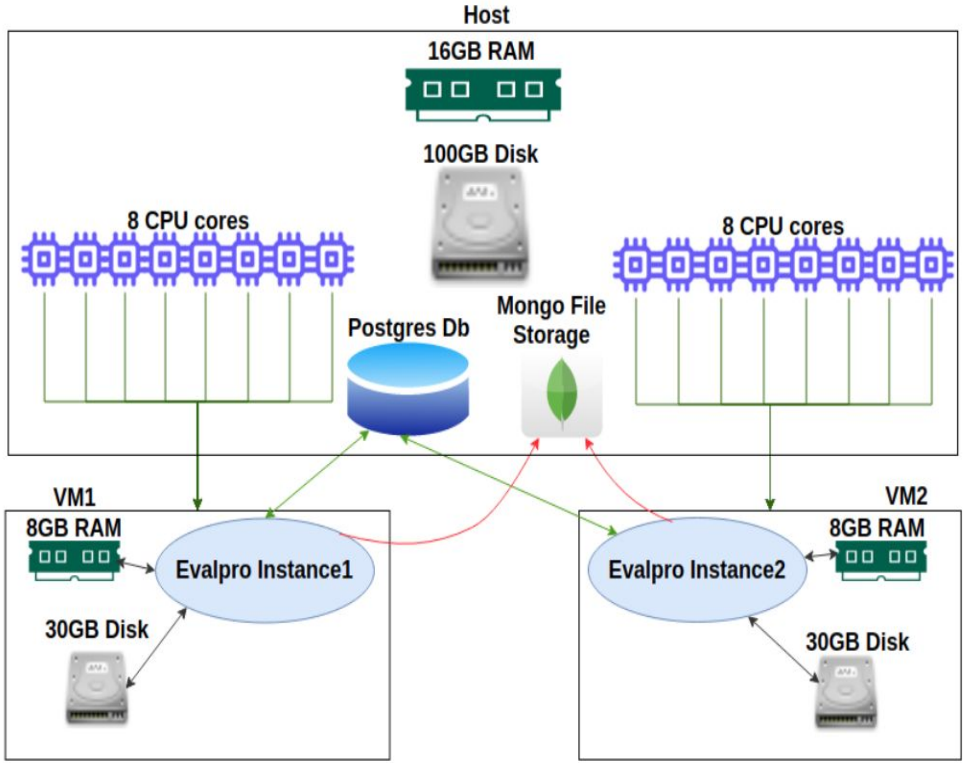
Evalpro updated architecture

Stores user uploaded files

File data is stored as stream of bytes



User data and files sharing VM setup with MongoDB

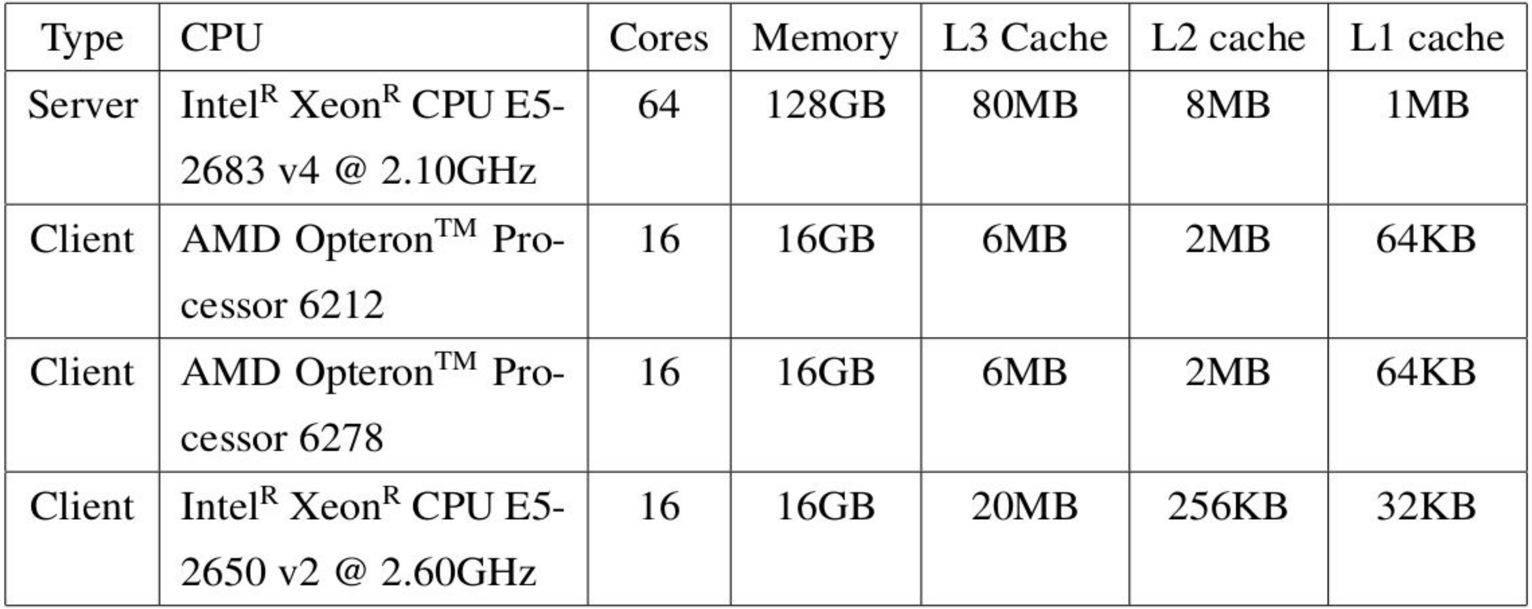


	Baseline	MongoDB for file storage
$Throughput_{max}(1)$ (req/sec)	0.3	0.29
$Throughput_{max}(16)$ (req/sec)	3.37	3.5
Ideal $Throughput_{max}(16)$ (req/sec)	4.8	4.64
$S(16)$	11.23	12

Hypothesis:
With large number of CPU cores throughput scalability will further increase

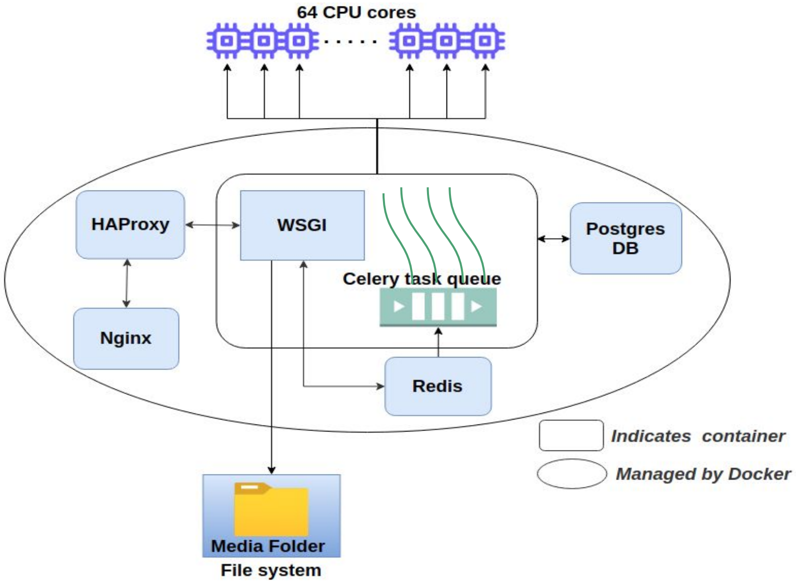
Throughput scalability slightly increased

Experiment setup using 64 CPU cores

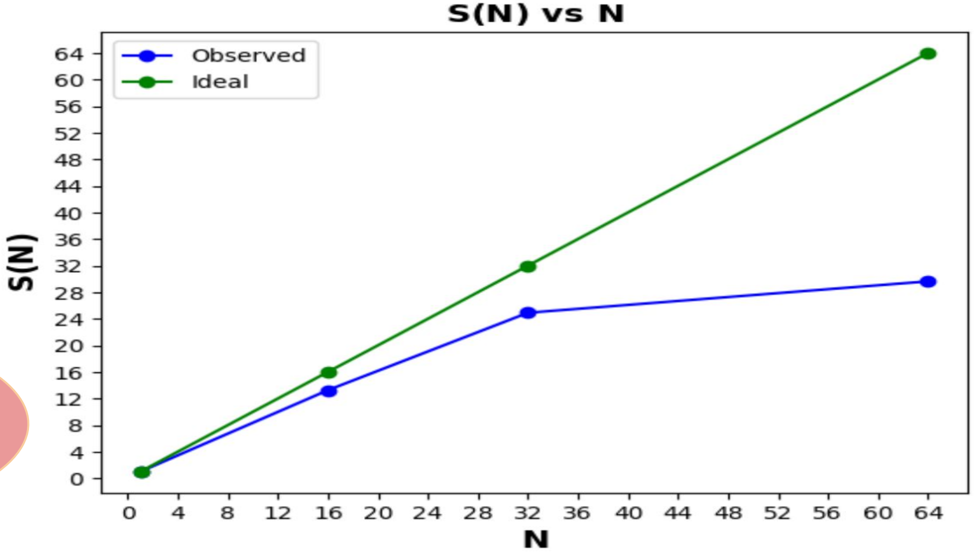


Type	CPU	Cores	Memory	L3 Cache	L2 cache	L1 cache
Server	Intel^R Xeon^R CPU E5-2683 v4 @ 2.10GHz	64	128GB	80MB	8MB	1MB
Client	AMD Opteron^{TM} Processor 6212	16	16GB	6MB	2MB	64KB
Client	AMD Opteron^{TM} Processor 6278	16	16GB	6MB	2MB	64KB
Client	Intel^R Xeon^R CPU E5-2650 v2 @ 2.60GHz	16	16GB	20MB	256KB	32KB

Baseline-64



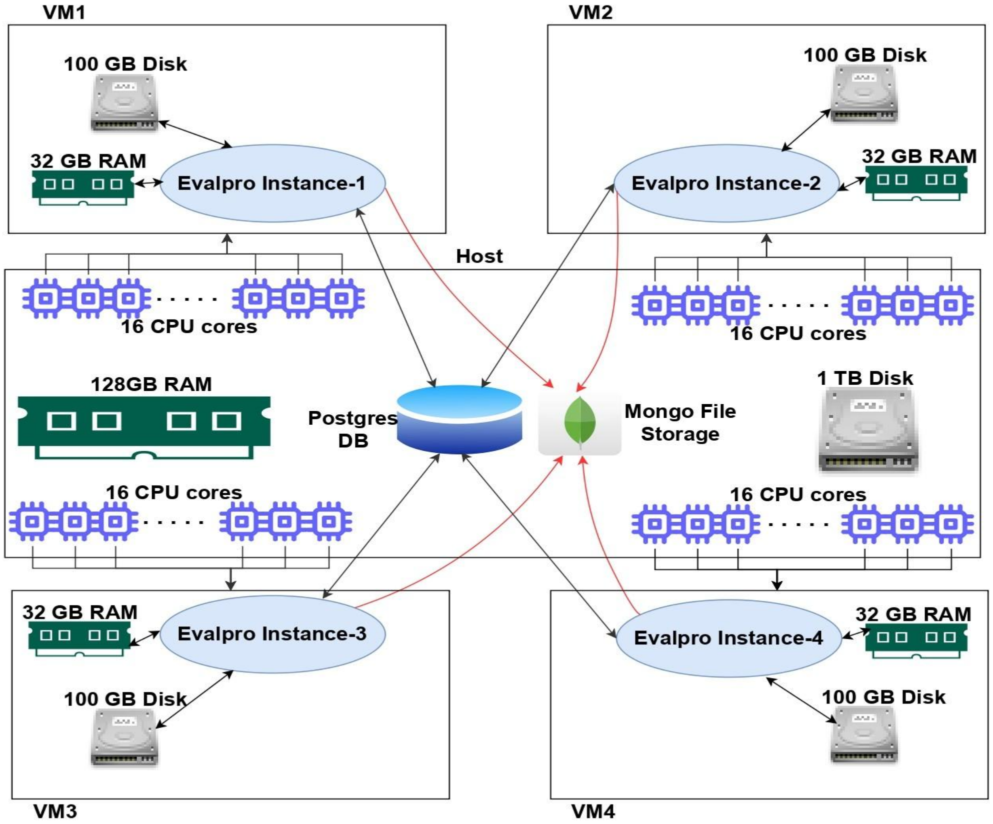
Throughput$_{max}$*(1)*	0.66 req/sec
Throughput$_{max}$*(16)*	8.75 req/sec
Throughput$_{max}$*(64)*	19.56 req/sec
Ideal *Throughput*$_{max}$*(64)*	42 req/sec
S(64)	29.6



All CPU cores 90-100% utilized

Linear scaling of throughput not achieved

User data and files sharing VM setup with MongoDB - 64

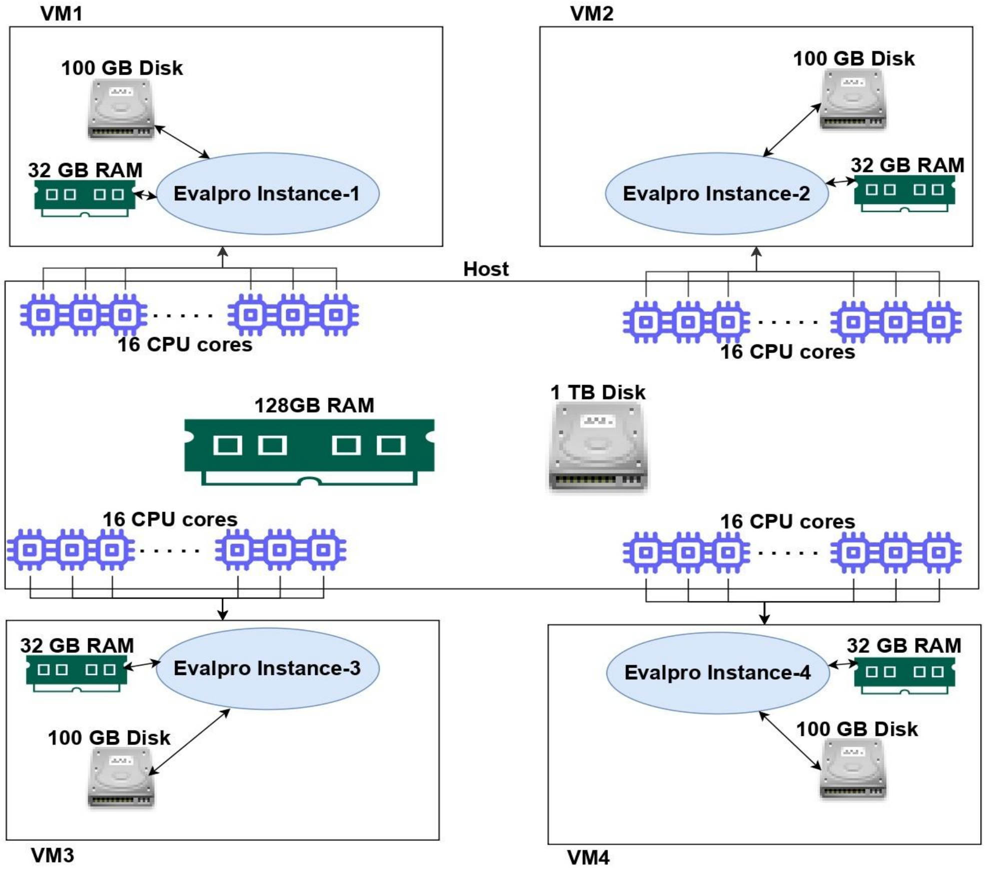


	Baseline-64	MongoDB for file storage - 64
Throughput$_{max}$(1) (req/sec)	0.66	0.47
Throughput$_{max}$(64) (req/sec)	19.56	11.27
Ideal *Throughput$_{max}$(64)* (req/sec)	42	30
S(64)	29.6	24

Throughput scalability decreased

Our hypothesis is wrong

Completely Isolated VM setup - 64



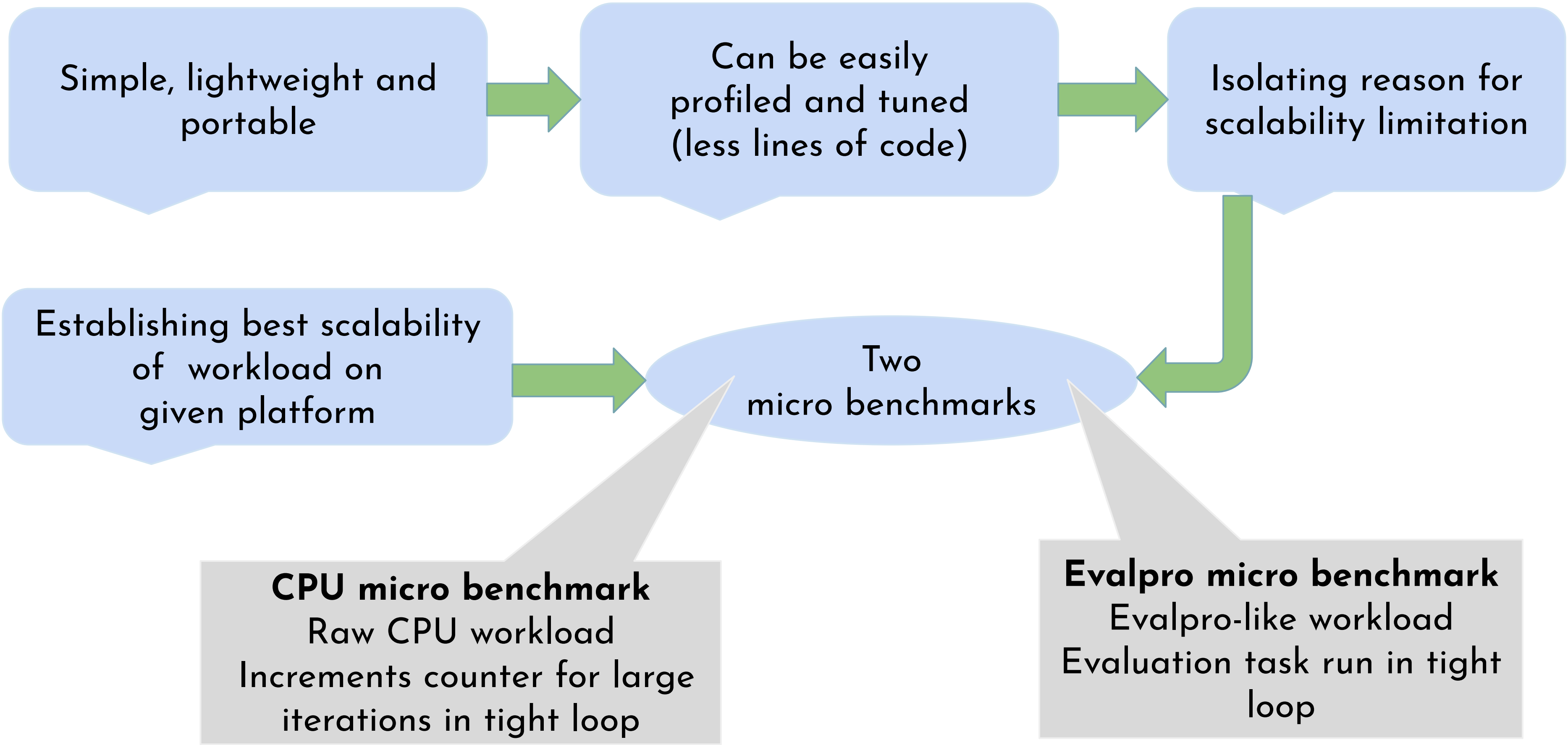
	Baseline-64	Isolated setup- 64
Throughput$_{max}$*(1)* (req/sec)	0.66	0.66
Throughput$_{max}$*(64* *)* (req/sec)	19.56	15.03
Ideal *Throughput*$_{max}$*(64* *)* (req/sec)	42	42
	29.6	23

With higher number of cores, even this setup didn't improve throughput scalability

Throughput scalability decreased

Micro benchmark Experiments

Need for micro benchmarks



Simple, lightweight and portable

Can be easily profiled and tuned (less lines of code)

Isolating reason for scalability limitation

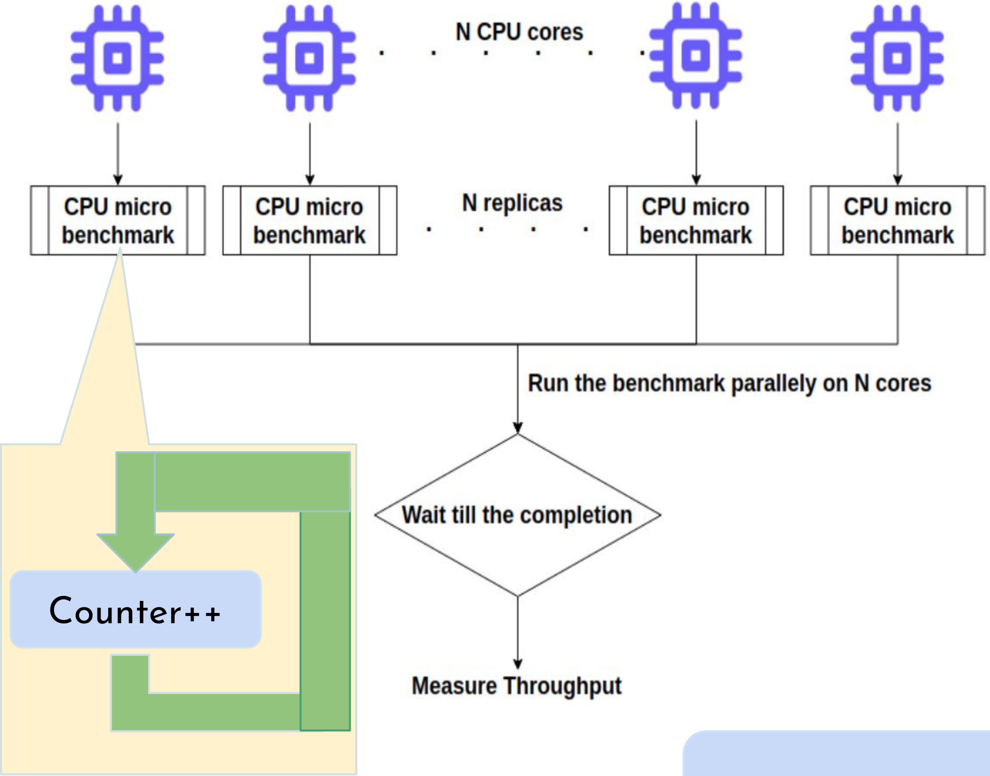
Establishing best scalability of workload on given platform

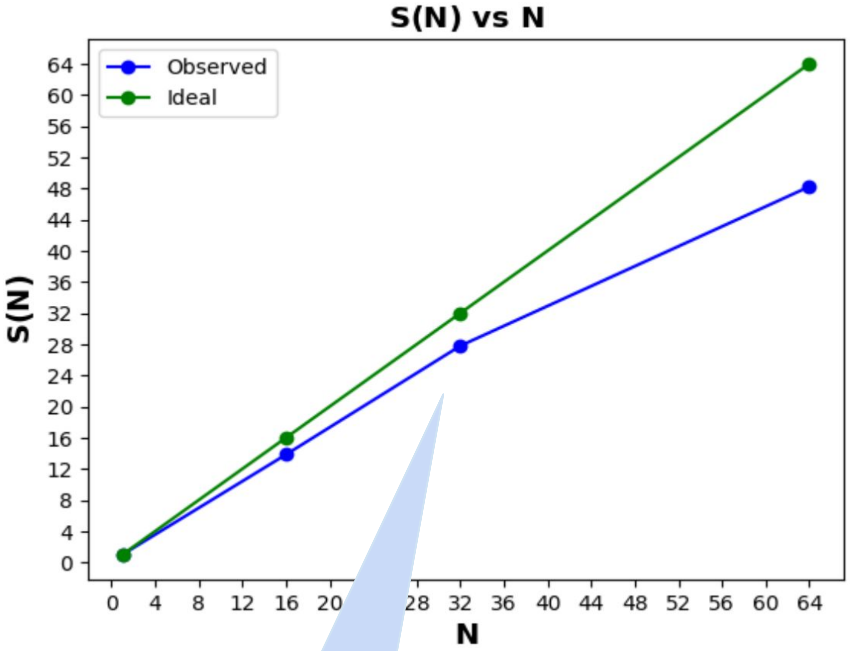
Two micro benchmarks

CPU micro benchmark
Raw CPU workload
Increments counter for large iterations in tight loop

Evalpro micro benchmark
Evalpro-like workload
Evaluation task run in tight loop

Experiments on CPU micro benchmark

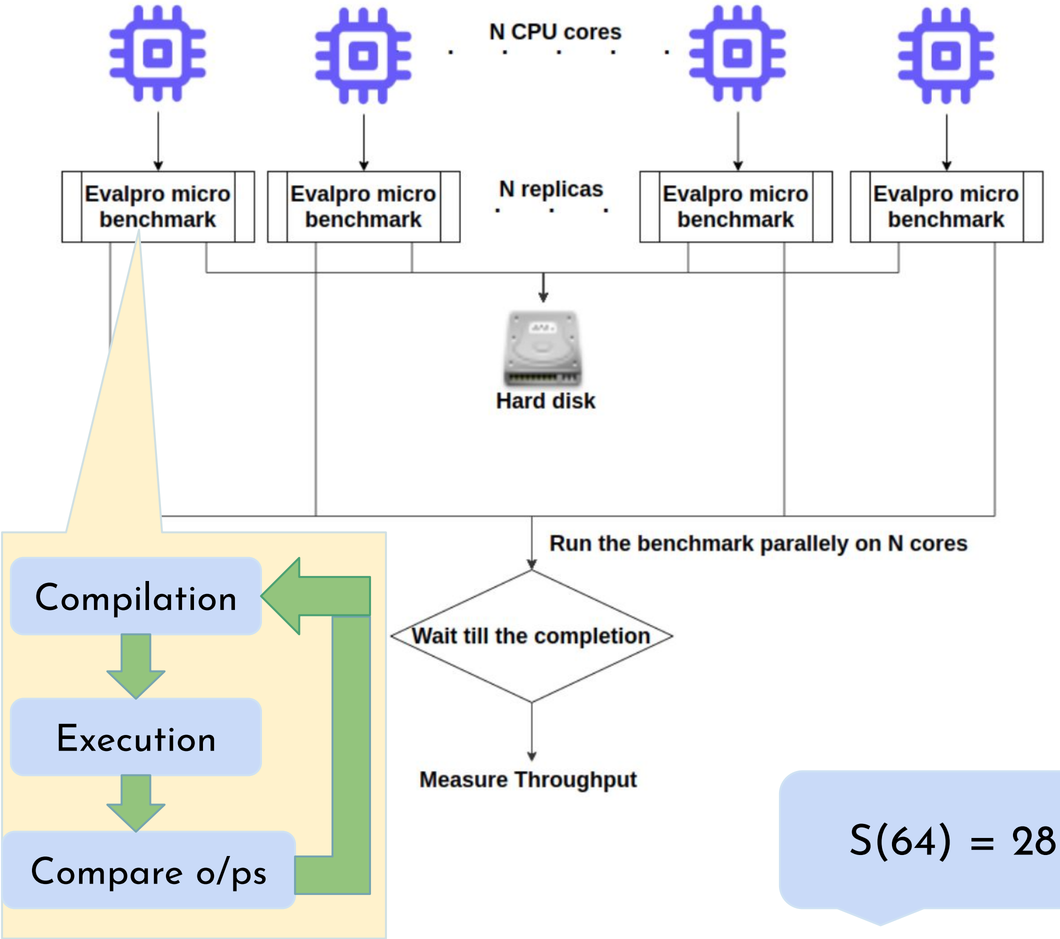




S(64) = 48

Throughput scalability is close to linear

Experiments on Evalpro micro benchmark



N CPU cores

Evalpro micro benchmark

Evalpro micro benchmark

N replicas

Evalpro micro benchmark

Evalpro micro benchmark

Hard disk

Run the benchmark parallely on N cores

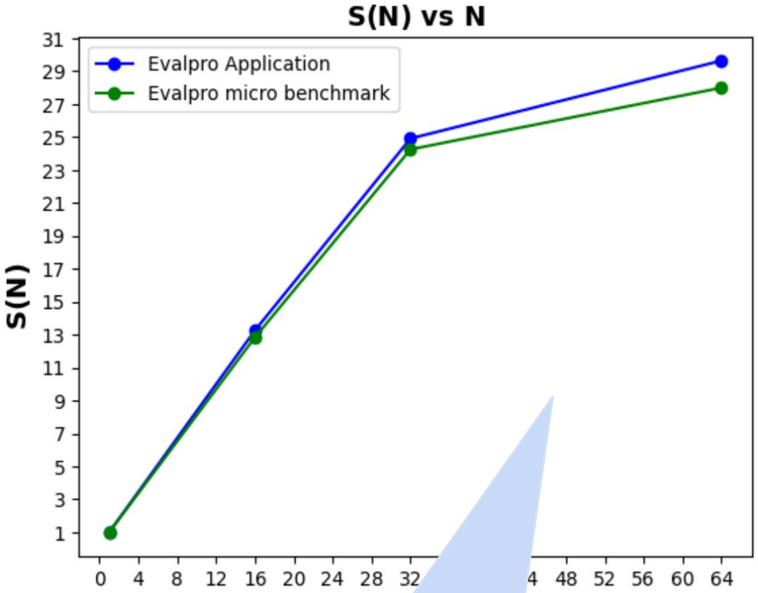
Wait till the completion

Measure Throughput

Compilation

Execution

Compare o/ps

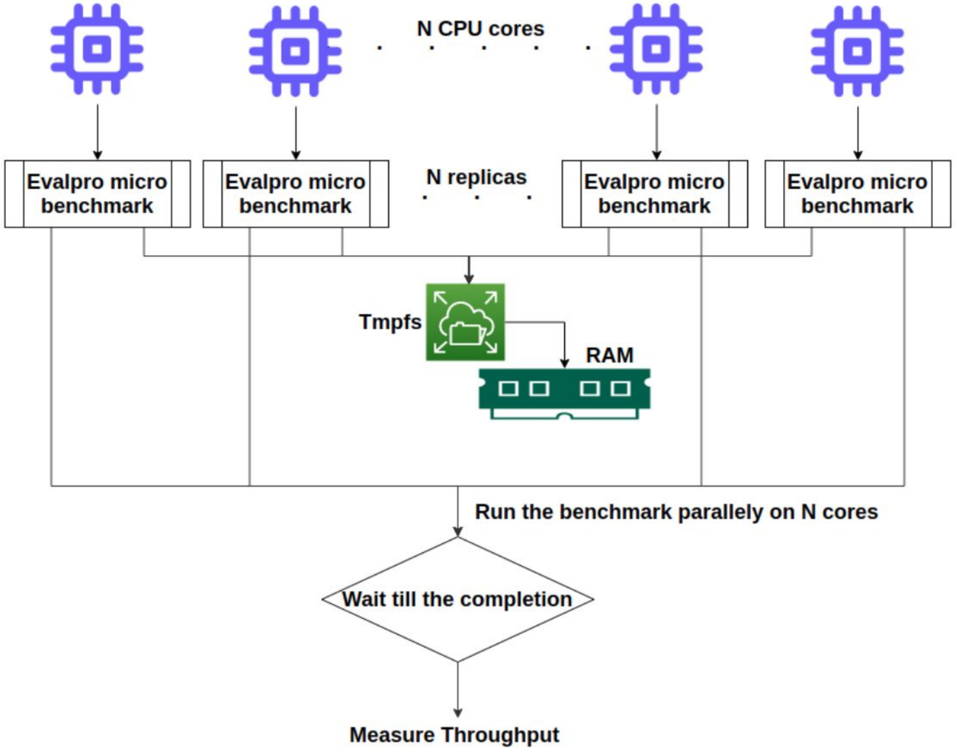


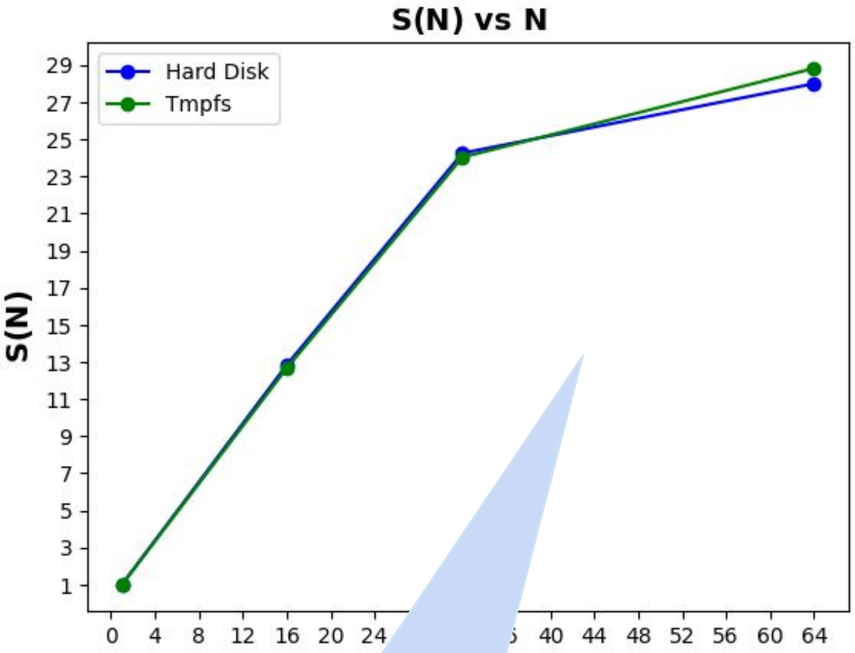
S(64) = 28

Throughput scalability is almost same as Evalpro application

Micro benchmark Bottleneck Analysis

Using Tmpfs in place of Hard disk





Using Tmpfs didn't improve throughput scalability

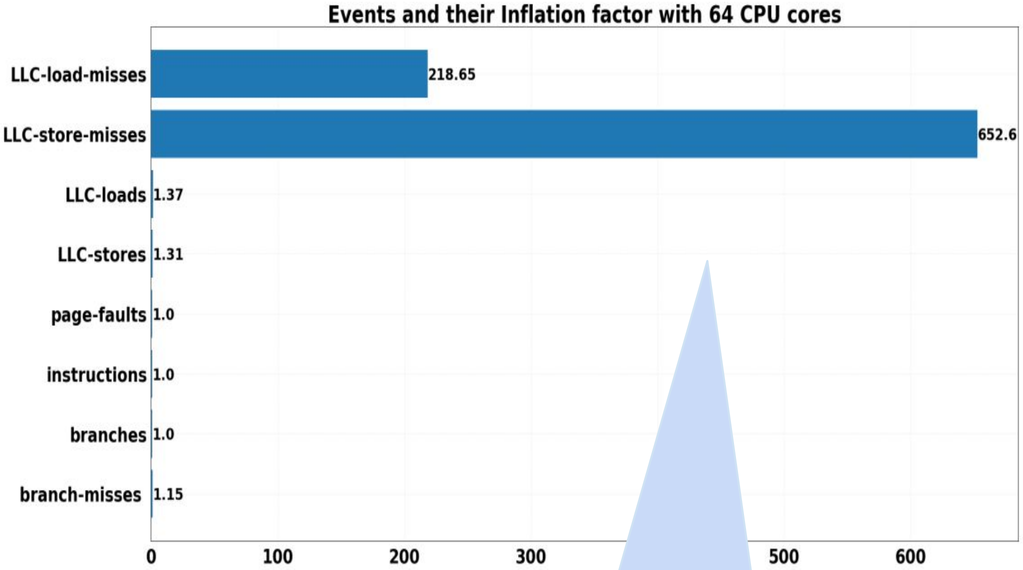
PERF analysis on Evalpro micro benchmark

PERF profiles
different software, hardware
events

$Count_{event}(N)$: Occurrence count of event with N cores
$Inflation\_factor_{event}(N)$: Inflation factor of event with N cores

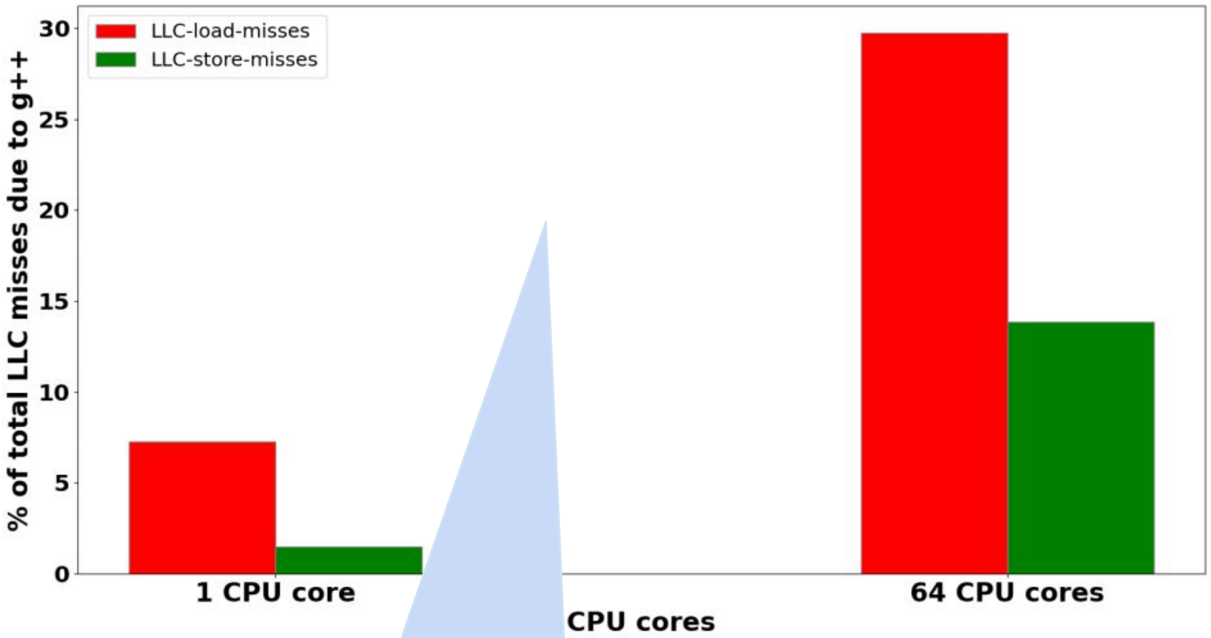
$$Inflation\_factor_{event}(N) = Count_{event}(N) \div (N \times Count_{event}(1))$$

Ideal $Inflation\_factor_{event}(N) = 1$


Events and their Inflation factor with 64 CPU cores

LLC load and store misses, disproportionately inflated with 64 CPU cores

PERF analysis at Program level

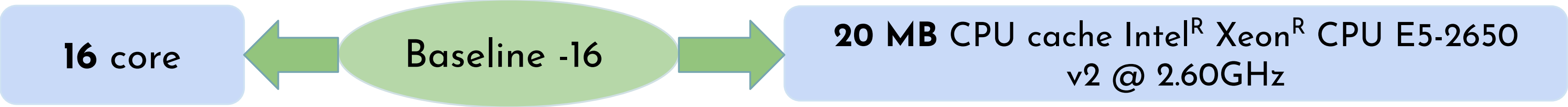


Hypothesis
CPU cache
is reason for
scalability limitation
of Evalpro application

LLC load and store misses increased drastically by g++ when number of cores is high

CPU cache is the reason for scalability limitation of Evalpro micro benchmark

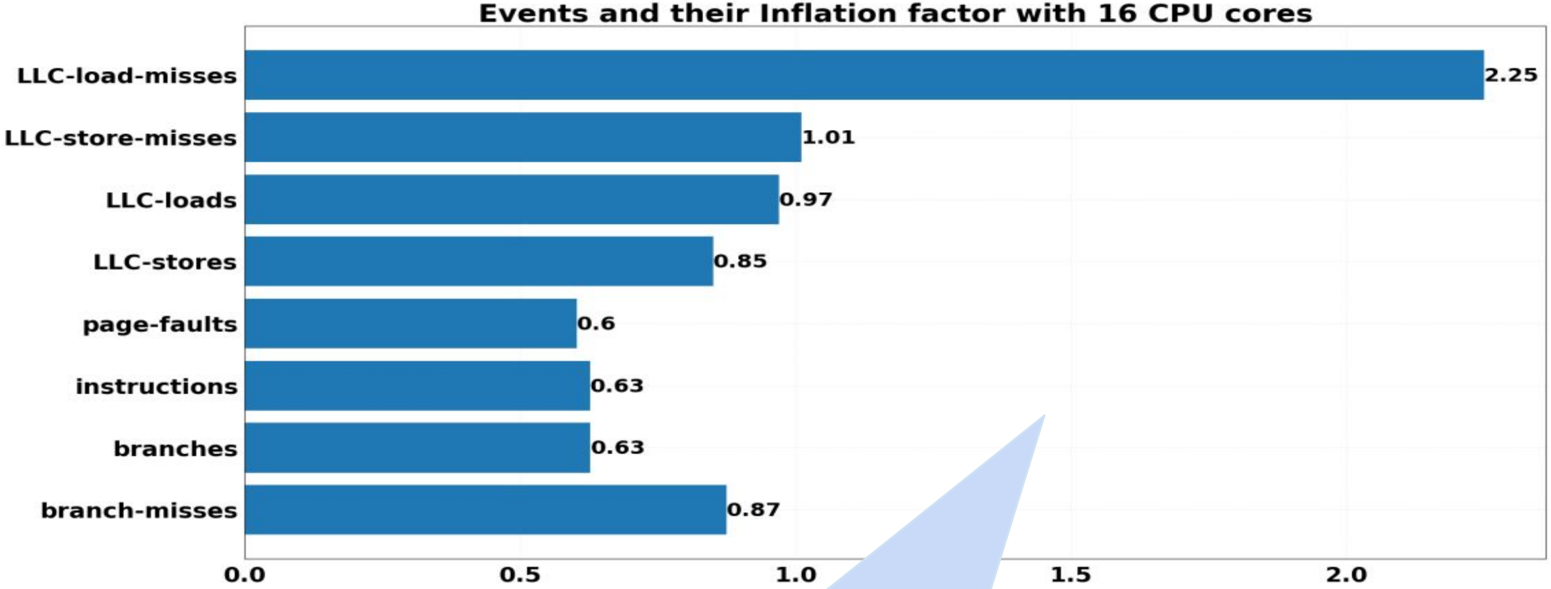
PERF analysis on Evalpro application



16 core

Baseline -16

20 MB CPU cache Intel^R Xeon^R CPU E5-2650 v2 @ 2.60GHz

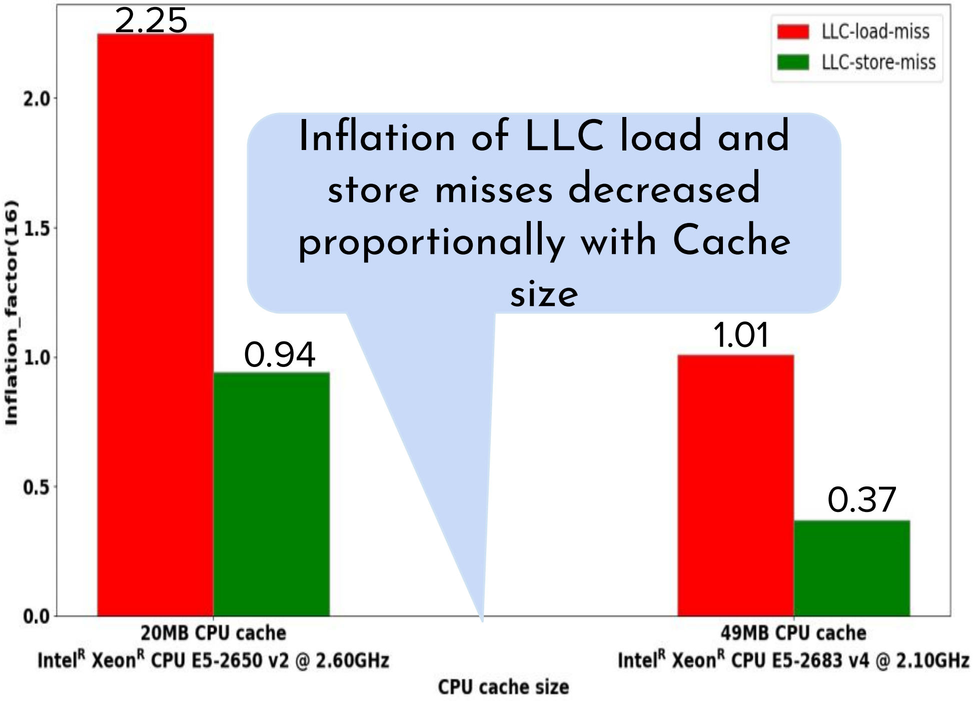


LLC load misses inflated
LLC store misses slightly inflated

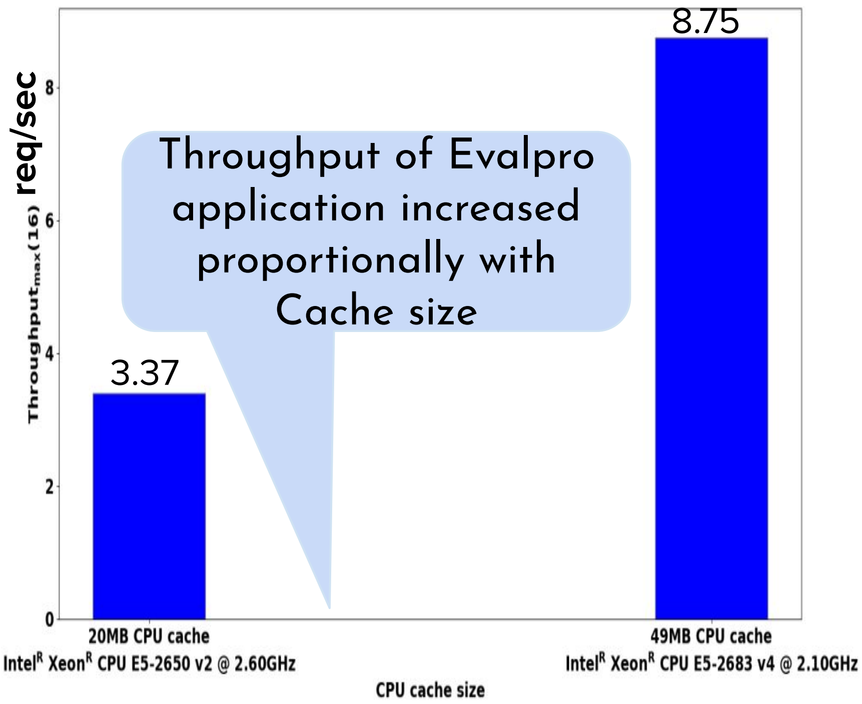
Cache size affect on Evalpro application

Baseline - 16
16 cores
20 MB cache

Baseline -64
only 16 cores enabled
89 MB cache



Inflation of LLC load and store misses decreased proportionally with Cache size



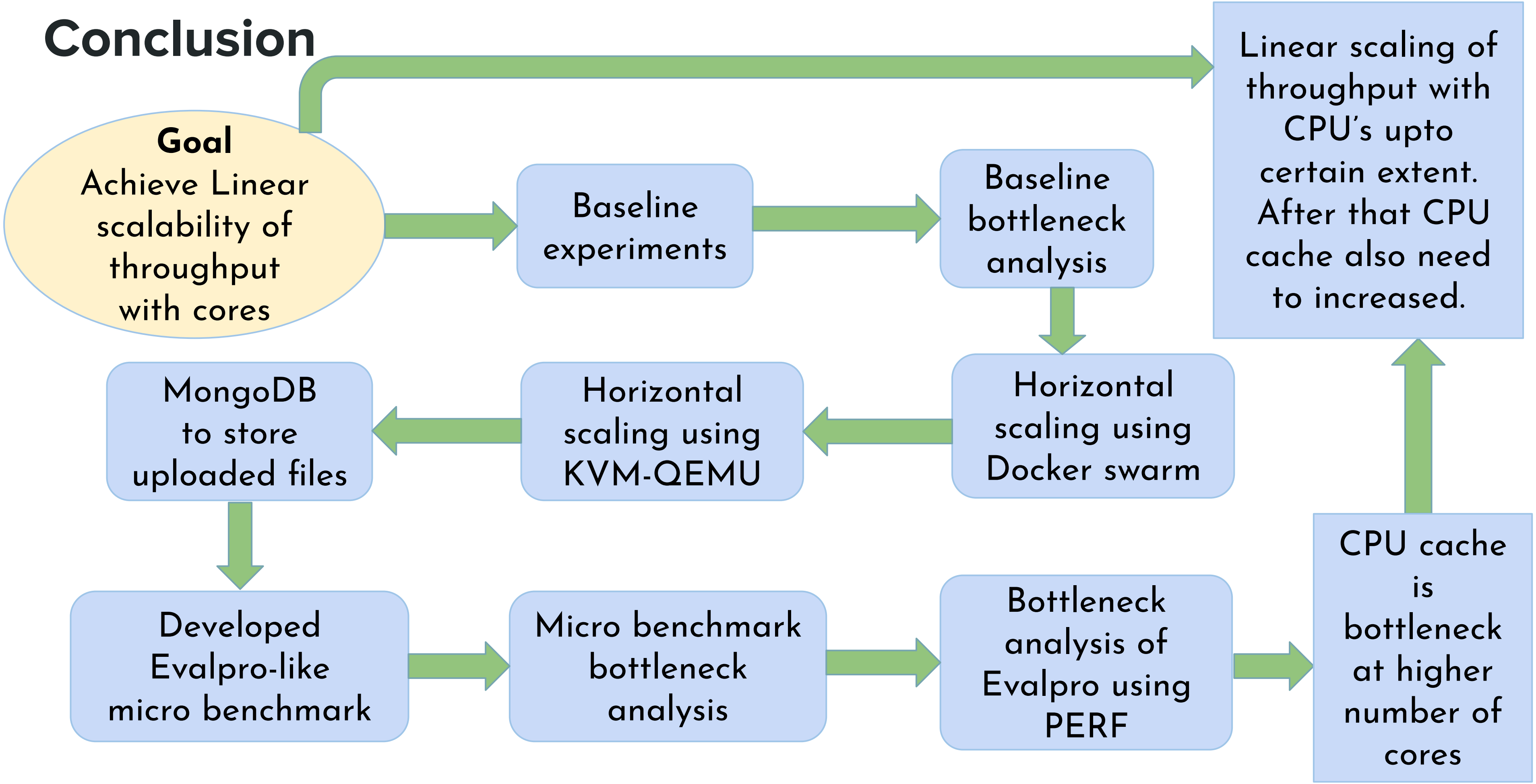
Throughput of Evalpro application increased proportionally with Cache size

Cache size affect summary

Our hypothesis
is correct
i.e CPU Cache
Is the bottleneck

	Baseline-16	Baseline-64	% of increase or decrease
CPU cache size for 16 CPU cores	20 MB	49 MB	+ 145%
$\text{Inflation\_factor}_{\text{LLC-load-misses}}(16)$	2.25	0.94	- 140%
$\text{Inflation\_factor}_{\text{LLC-store-misses}}(16)$	1.01	0.37	- 170%
$\text{Throughput}_{\text{max}}(16)$	3.37 req/sec	8.75 req/sec	+ 160%

Conclusion



Goal
Achieve Linear scalability of throughput with cores

Baseline experiments

Baseline bottleneck analysis

Horizontal scaling using Docker swarm

Horizontal scaling using KVM-QEMU

MongoDB to store uploaded files

Developed Evalpro-like micro benchmark

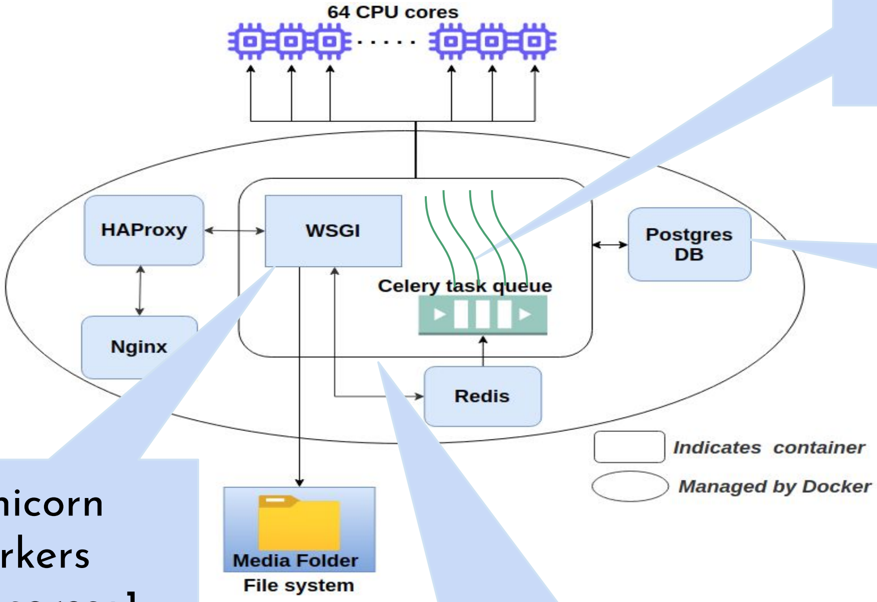
Micro benchmark bottleneck analysis

Bottleneck analysis of Evalpro using PERF

CPU cache is bottleneck at higher number of cores

Linear scaling of throughput with CPU's upto certain extent. After that CPU cache also need to increased.

Final Recommendation for Scaling EvalPro



celery threads at least number of cores

postgres connections to large number i.e around 100000

Gunicorn workers = 2*ncores+1

No need for multiple WSGI+Celery replicas with above configurations

Migrate Evalpro to cloud

Pay for use

Easily scale performance linearly with CPU cores and cache size

Future work

Scalability of compilation workloads

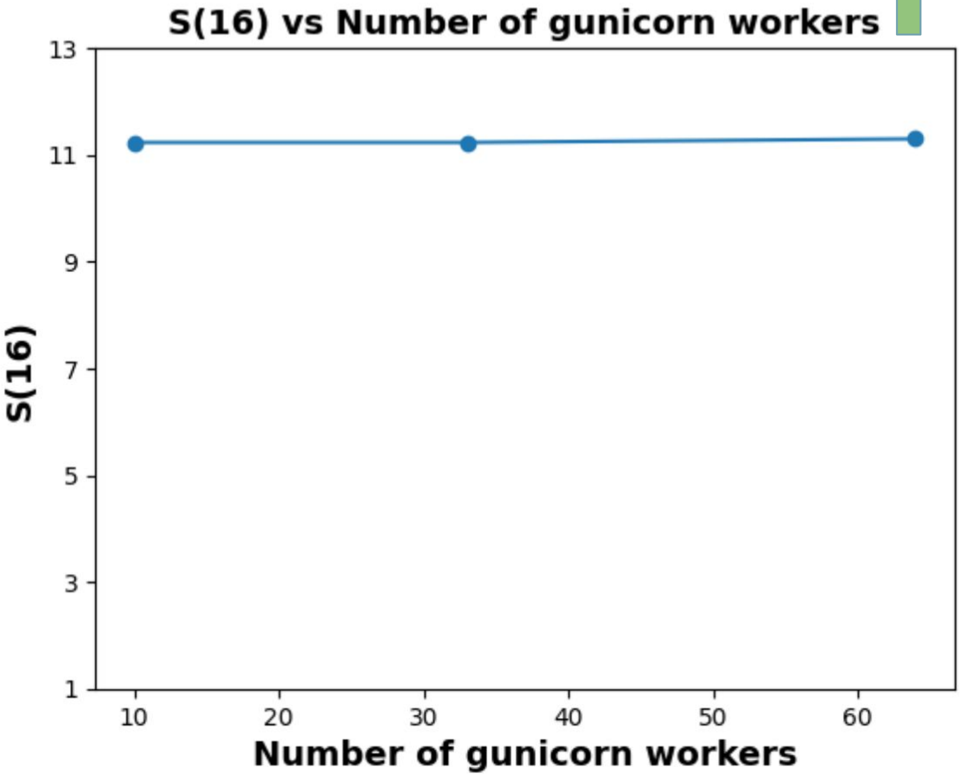
Developing a browser plugin to do compilation at client side

Improving single core throughput of Evalpro

THANK YOU

Backup Slides

Increasing gunicorn workers



Increasing gunicorn workers didn't improve throughput scalability

Gunicorn workers are not bottleneck

Gunicorn workers are WSGI worker threads

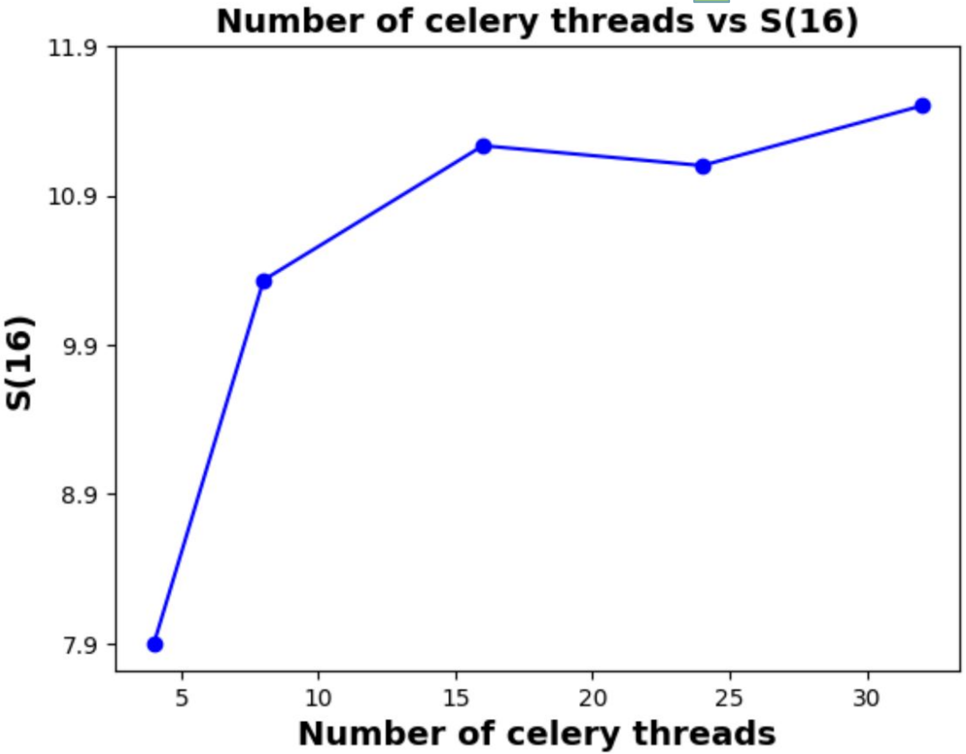
Baseline gunicorn workers = 10

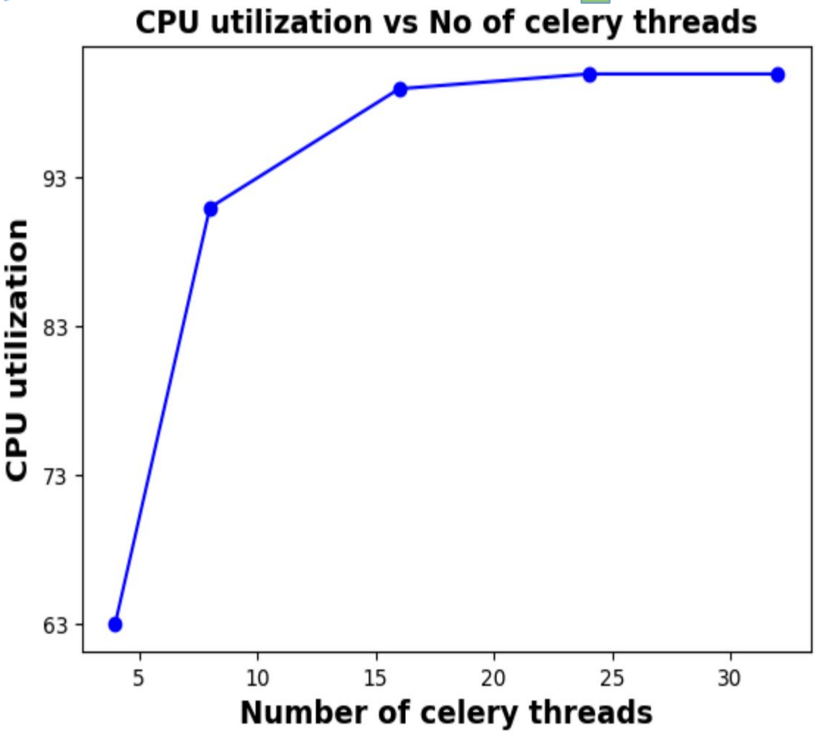
Gunicorn documentation says,
Ideal value = 2 x ncores +1

Tuning celery threads

Auto grading tasks processed asynchronously

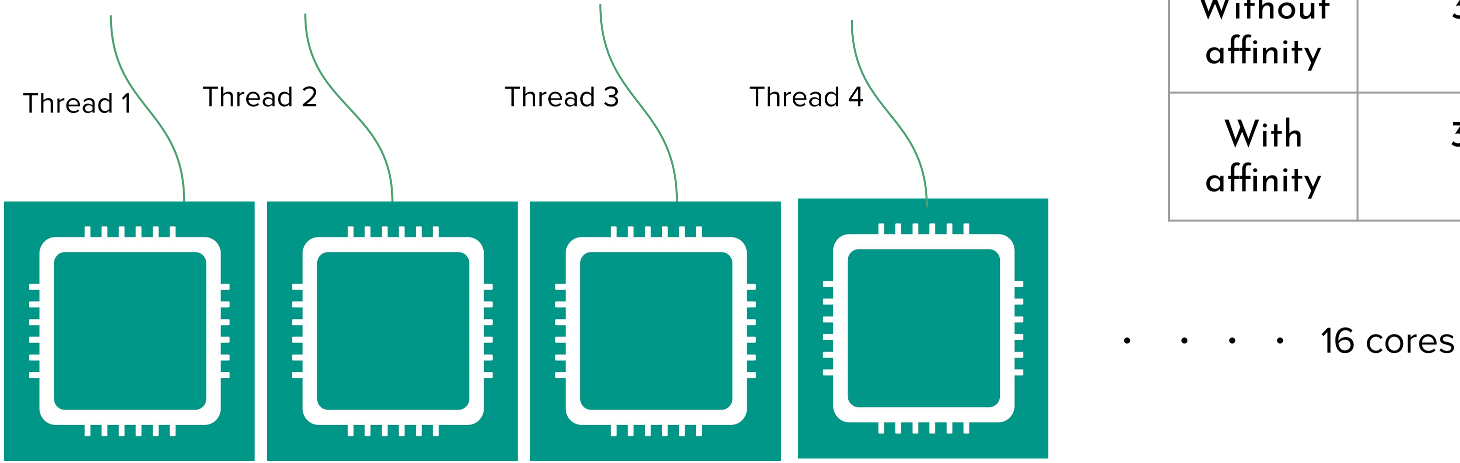
Celery threads are bottleneck when less than number of CPU cores





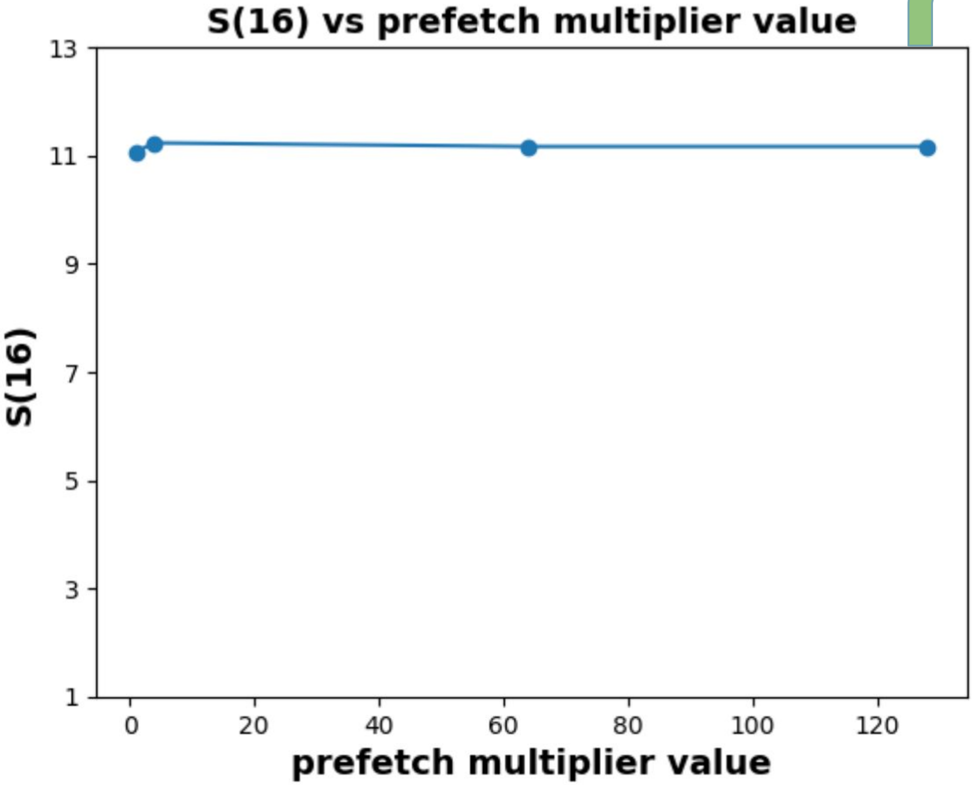
Setting CPU affinities for celery

Type	Throughput (req/sec)	S(16)
Without affinity	3.37	11.23
With affinity	3.32	11



Setting CPU affinities didn't improve throughput scalability

Increasing celery prefetch multiplier



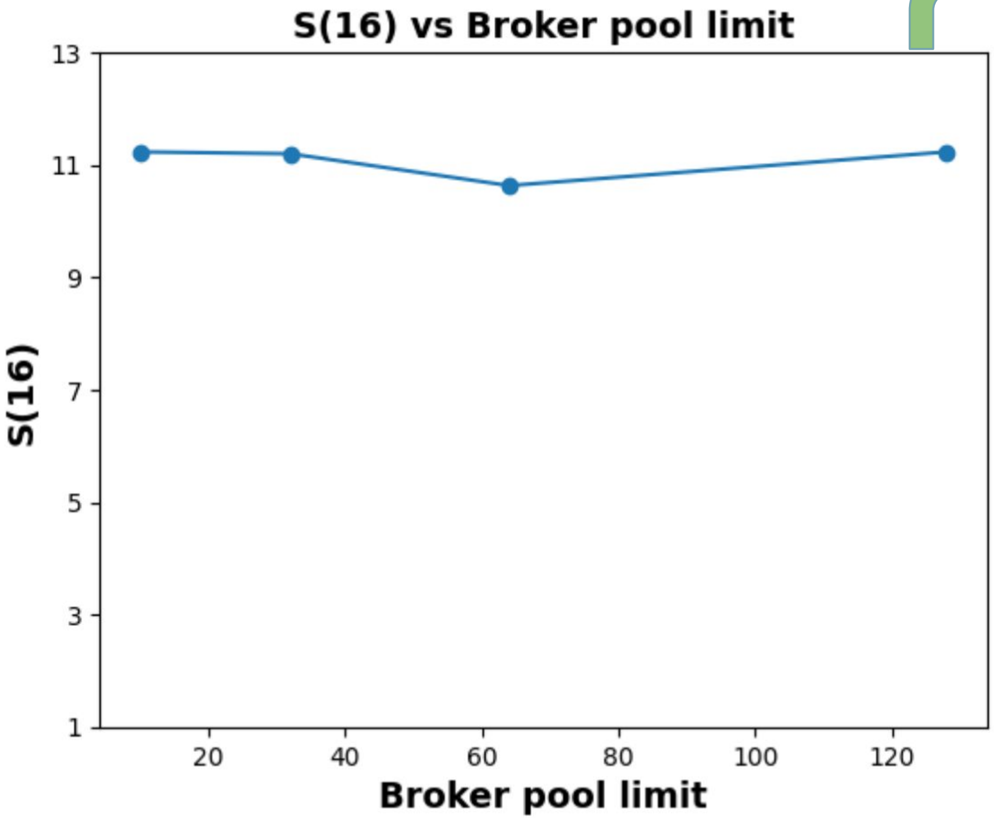
Increasing prefetch multiplier didn't improve throughput scalability

No of tasks celery picks and keeps in memory at a time

Baseline prefetch multiplier value = 4

Celery Prefetch multiplier is not bottleneck

Increasing celery broker pool limit



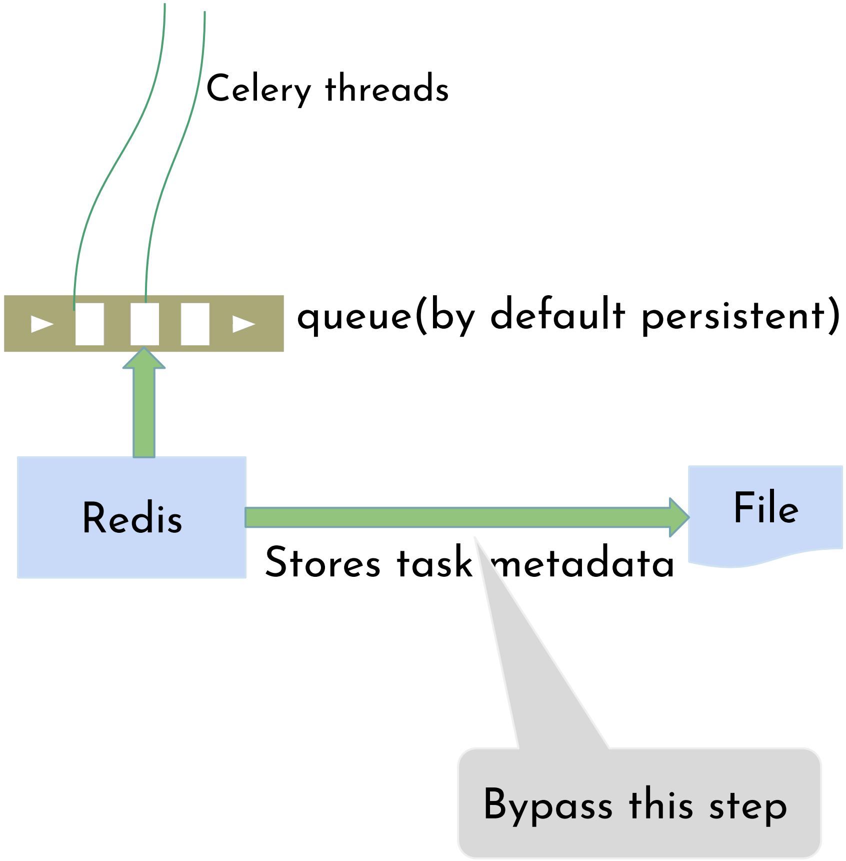
Increasing broker pool limit didn't improve throughput scalability

No of connections kept open for the broker

Baseline broker pool limit = 10

Celery Broker pool limit is not bottleneck

Using transient celery queue



Celery threads

queue(by default persistent)

Redis

File

Stores task metadata

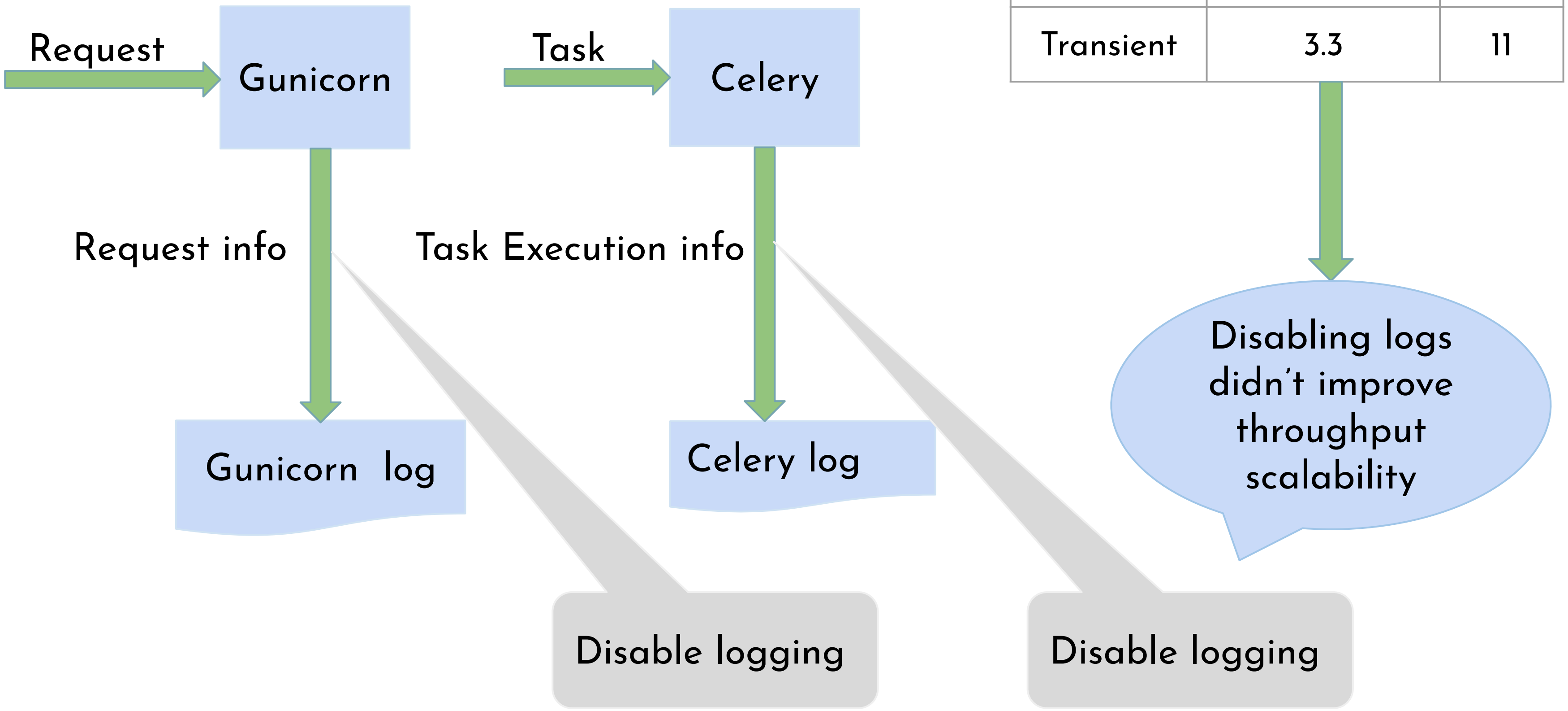
Bypass this step

Type	Throughput (req/sec)	S(16)
Persistent	3.37	11.23
Transient	3.3	11

Making celery queue transient didn't improve throughput scalability

Disabling writing to log files

Type	Throughput (req/sec)	S(16)
Persistent	3.37	11.23
Transient	3.3	11



Disabling logs didn’t improve throughput scalability