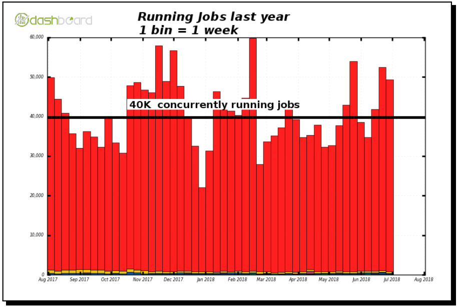
# Improving efficiency of analysis jobs in CMS.

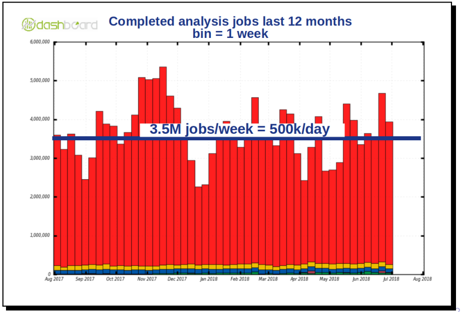
Todor Ivanov for the CMS collaboration

July 10, 2018

## Introduction

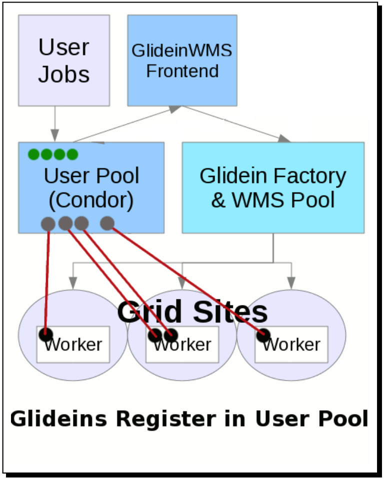
- CMS experiment has a workload management system that schedules and executes MonteCarlo production and user Analysis tasks in a distributed Grid infrastructure.
- **Past focus :** make jobs run, offer users **painless and transparent** access to the Grid. We have been largely successful.
- **Recent focus:** on **efficiency and optimisation:** turnaround time, CPU efficiency, scalability of the system.
- **This contribution:**
  - improving the execution of analysis jobs. I.e. job submitted by users (few hundred different people using the system at any given time)
  - thus **w/o access** to the application itself



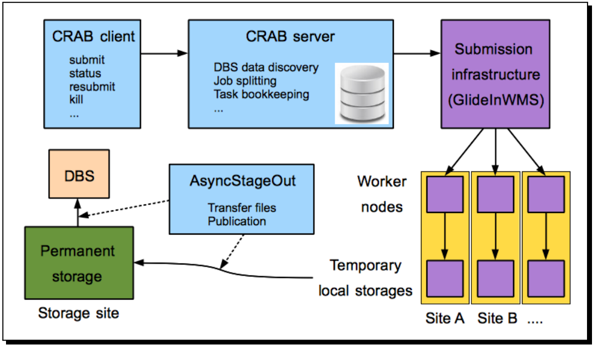
## Global Pool & glideinWMS

- Based on glideinWms:
  - **Users:** Vanilla HTCondor jobs via **ad hoc tools : WMA** for production, **CRAB** for analysis
  - **Glidein FrontEnd:** glideins (**PilotJobs**) → Grid Sites
  - **PilotJobs: 48 hours; 8 cores;**
  - **1 PilotJob** → **1 HTCondor** *startd* which joins the *GlobalPool*
  - **1 PilotJob** runs **many multi/single-core jobs** and keeps reallocating freed up cores until the end of its lifetime
- **CMS takes ownership** of all issues of pool **fragmentation** due to running variable number of multi/single-core jobs of different length



# Analysis jobs submission: CRAB

- **CMS Remote Analysis Builder (CRAB):**
  - Turns a high level request *(run this executable on this set of files)* into a set of jobs whose execution is controlled by HTCondor DAGMAN
  - Later an *Asynchronous StageOut* component moves job outputs from remote site storage to the user preferred site
    - Optionally record the files in CMS *Dataset Bookkeeping System*
- **Splitting:** 1 request (task) $\rightarrow$ many jobs

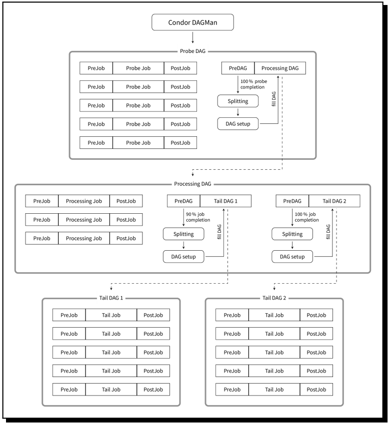


## Three lines of work

- **Automatic Splitting**
  - Optimise **job running time** (splitting a large task in many jobs)
    - *Too long:* High chance of killing by glitch → wasted resource
    - *Too short:* Too many jobs, unnecessary load on infrastructure, too much time in overheads
- **Time Tuning**
  - Optimise **job to slot allocation** (tune the job time requirement)
    - Avoid killing/restarting pilots too soon, exploit the tail of each slot
    - Majority of jobs ask for 20h but only run 30min or less, how close to the pilot end of life is OK to start them ?
- **Overflow**
  - Optimise **scheduling of jobs across sites** (overflow from busy site queues)
    - We used to run where data are
    - We can now exploit **xrootd** to run also at other sites
    - But can't fully ignore where data are

## Automating Splitting: Theory

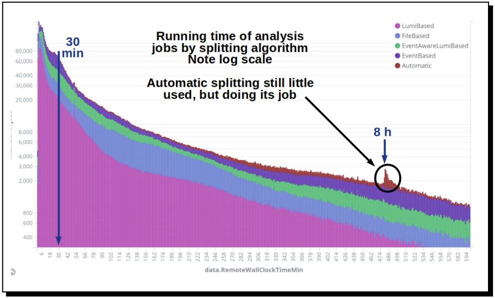
- **Before:** 1 task → 1 DAG
  - Splitting parameters **configured by the user**
  - Results in **thousands of very short jobs**
    - Bad for scaling
- **After:** 1 task → a few DAG's
  - All decisions taken **out of user hands**
  - One PROBE DAG to estimate **time, memory, disk needs**
  - Splitting parameters **computed in per event basis**
  - Target: **8h jobs**
  - One PROCESSING DAG to do the work
  - Three tail stages - 3 TAIL DAGS
    - one when 50% of the PROCESSING is done,
    - one at 80%,
    - one up to the end
  - Fewer TAILs for small tasks (<100 jobs)
  - Jobs are set to run for a **fixed time**. If they don't complete all work, they finish **gracefully** and the remaining work is taken care by the tail jobs

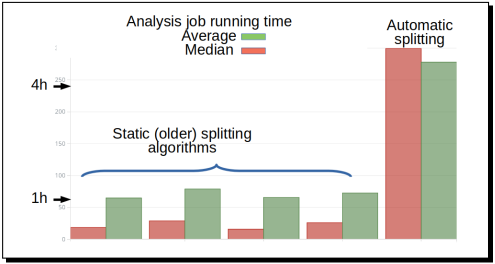


## Automating Splitting: Practice

- **Before:** 1 task → 1 DAG
  - **Good:** Splitting done in the TaskWorker (**one server**, centralized logs, **easy to debug**)
  - **Bad:** User finds best splitting by trial and error running same things **N times (invisible waste)**
  - **Overall:** Non optimal, but when things go wrong we blame the user and life goes on
- **After:** 1 task → a few DAG's
  - **Good:** **It really works**
  - **Bad:** Splitting done on the schedd (**15 machines**, log scattered in user directories, hard to rerun in debug mode).
  - **Hard part:** **Not all use cases** can be addressed, e.g. for MonteCarlo generation there is no splitting.
  - **Overall:**
    - When things go wrong, it is on us to explain, solve and prevent
    - Large variation of WN CPU power and data serving performance at sites introduces large uncertainty in jobs run times. Leads directly to the need for Time Tuning (next slides).

# Automating Splitting: Results



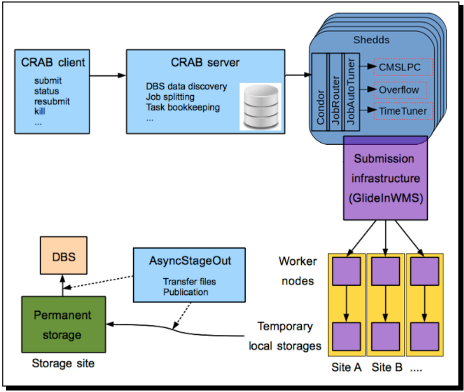




- In production since February 2018.
- Users encouraged but not pushed.
- Few issues, generally high satisfaction.
- Extending usage requires education campaign: manpower issue.
  - Next step once all commissioning work is completed.
- **Current adoption is 2% but could grow to >> 50%**

## Automating Tuning and CRAB3:

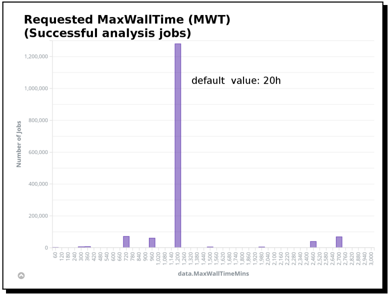
**Editing job requirements: HTCondor JobRouter**
Both following lines of work (Time Tuning and Overflow) rely on modifying job requirements while jobs are idle in the HTCondor queue → different scheduling → freedom to optimise.

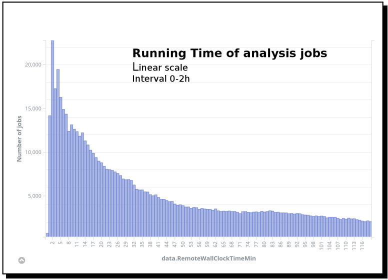
- **There's a tension:**
  - **Global overview** (best decisions) vs **Local action** in each schedd (efficient).
- **And there's a criticality:**
  - **Spiral of Death:** Massive condor_qedit ⟹ schedd load ⟹ long negotiator time ⟹ starving pilots ⟹ job restarts ⟹ more load on the schedds
- **Our solution:**
  - A central process to collect information and make **stats based on a feed of HTC classAds to Elastic Search**.
  - HTCondor **JobRouter** to do the actual classAd remapping locally to each schedd
  - Strategy already in use for organized Production, but **larger workflows** and much more **top/down control**
- **Slow feedback** → care in turning knobs
  - $O(10min)$ in what we do - $O(hours)$ in HTCondor reaction



## Time Tuning: Theory

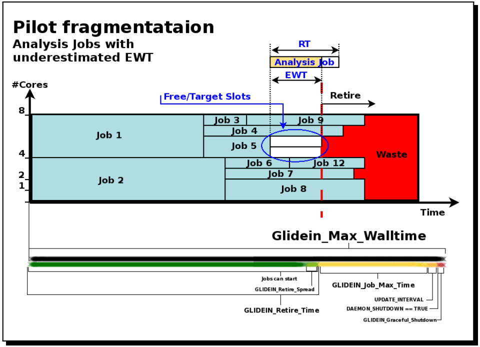
- **Jobs request** a **MaxWallTime (MWT)** at submission. HTCondor **kills** jobs which hit it.
  - MWT is a common classAd atribute for all jobs in a task.
- **Problem:** Majority of jobs ask for the **default 20h** MWT but most only run **30min** or less
  - This is not because users are nasty
    - Even if jobs are created equal, they run for different times
    - Even good willing users need to indicate a largish, safe, value
  - Automatic splitting will help, but not all tasks will use it:
    - Set a realistic limit for PROCESSING DAG
    - Jobs which run longer are resplitted so that a safe maxTime can be set which still is $O(hour)$
- **Approach: Introduce EstimatedWallTime (EWT).**
  - Use EWT to schedule, MWT to kill
  - EWT : realistic. MWT: conservative. EWT $<<$ MWT



# Time Tuning: Practice

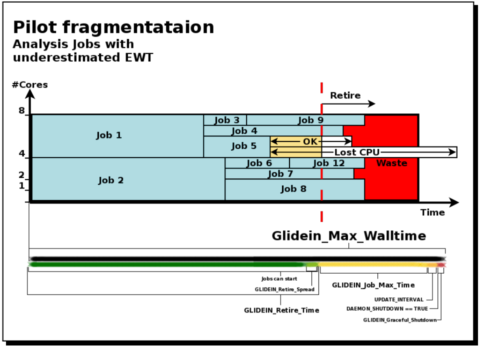
- **Implemented solution:**
  - EWT computed **as soon as one job completes** and dynamically updated every 10 min
  - EWT estimate algorithm tuned to contain most but not all jobs:
    - Pick **95th percentile** of collected RunTimes and apply **correction** dependent on **#jobs**
  - EWT added and updated in each job via JobRouter.
  - Jobs can **keep running** in the pilot's **tail**(the pilot's retire time) even after EWT expires, up to MWT
  - If a job reach a pilot's **end of lifetime** it is automatically and transparently **restarted** by HTCondor (but CPU is wasted)
-




- **Issues to overcome:**
  - Limited statistics to work with, first jobs to complete may be **not representative**
    - This is a bullet that we have to bite
  - Properly measure what we gain **(less fragmentation)** and what we lose **(wasted CPU)**

# Time Tuning: Practice

- **Implemented solution:**
  - EWT computed **as soon as one job completes** and dynamically updated every 10 min
  - EWT estimate algorithm tuned to contain most but not all jobs:
    - Pick **95th percentile** of collected RunTimes and apply **correction** dependent on **#jobs**
  - EWT added and updated in each job via JobRouter.
  - Jobs can **keep running** in the pilot's **tail**(the pilot's retire time) even after EWT expires, up to MWT
  - If a job reach a pilot's **end of lifetime** it is automatically and transparently **restarted** by HTCondor (but CPU is wasted)

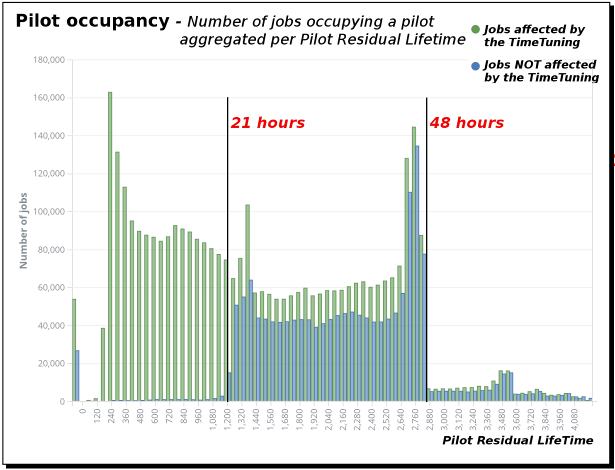
- 


- **Issues to overcome:**
  - Limited statistics to work with, first jobs to complete may be **not representative**
    - This is a bullet that we have to bite
  - Properly measure what we gain **(less fragmentation)** and what we lose **(wasted CPU)**

## Time Tuning: Results

### CONSERVATIVE SETUP:

| | |
|---|---|
| JOBS running less than EWT: | 95% |
| JOBS running longer but completed: | 4% |
| JOBS restarted once and completed: | 1% |



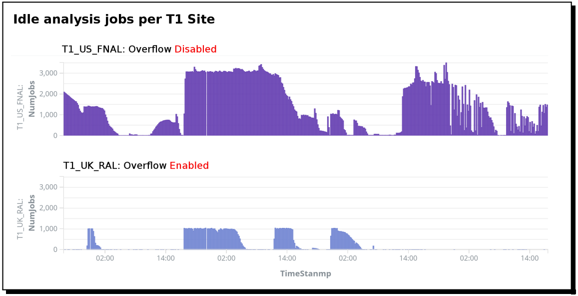

### STILL VERY EFFECTIVE:

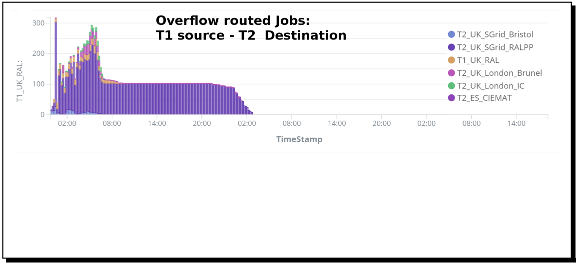
- Jobs which were TimeTuned filled mostly short living pilots
- Jobs which were not Time-Tuned filled pilots longer than 21 hours

## Overflow: Why, What, How

- **Problem:** By default jobs are sent to sites which hosts the input data $\Longrightarrow$ **long waiting times** when target sites are overloaded.
- **Solution:** run some of those jobs elsewhere. CMS software exploits our **xrootd data federation** for remote reads
- **Old Way** ad hoc glideinWms FrontEnd group to define topology and running limits. Deployed for US sites since a few years. Works, but can't extend it:
  - US sites large and homogeneous. Dedicated pilots $\rightarrow$ pool fragmentation. glideinWms suffers with many FrontEnd groups
- **New Way:** JobRouter **dynamically changes list of desired execution sites** for some jobs
  - Central overview opens to advanced **scheduling decisions** :e.g. add WAN information
- **Difficulties:**
  - will an overflow job complete earlier ? how much wait is too much wait ?
  - how much (more) remote reading a given site can handle ?
  - more remote reads = more,harder to debug, failures
  - if there are failures, are our remote reads the reason ?
  - large differences in site size and connectivity
  - need to go over country boundaries
- **Approach:** **Start slowly, watch carefully, push slowly, iterate**.
  - Users stand "wait but OK" better than "fail and need to retry"
  - Site admins do not like more problems and expect us to have extreme care

# Overflow: Results




- **Current use limited to T1s:** – Facing the Tier1 problem is Critical for us. Analysis jobs get a small share in T1s. But there are datasets that are currently placed only at T1
- **Work in progress:** Implementing a maximum overflow in a country and providing a way to substitute the old Overflow

## Summary

- We operate a complex setup with $O(40k)$ analysis jobs running at any moment and where many things change constantly outside CMS Analysis Operation control.
- We have to be careful. It is NOT easy to push changes in production transparently to the user community nor to disentangle effect of the various changes.
- And that while our monitoring infrastructure is being migrated/rebuilt.

**But we managed to deploy** the needed knobs and dials and we look forward to learn how to better tune the system.

## Future lines of work

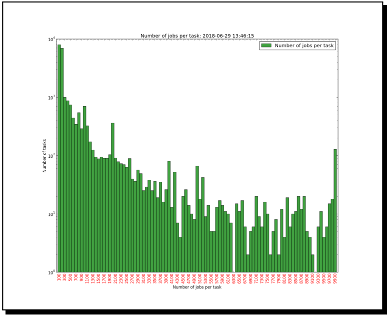
- Some of what we do requires guessing what users really need
  - Inferring the behaviour and needs of large tasks from small initial samples
  - Very tricky when we have many tasks with not many $(< 100)$ jobs each
- Some requires guessing the future
  - How sites and networks will react to load that we are about to place on them
- Will be a good arena for:
  - Central vs. Local control
  - Infer large sample behaviour from limited statistics
  - Machine Learning
  - Network scheduling
- **Future will be more fun than the past !**

**Thank You!**

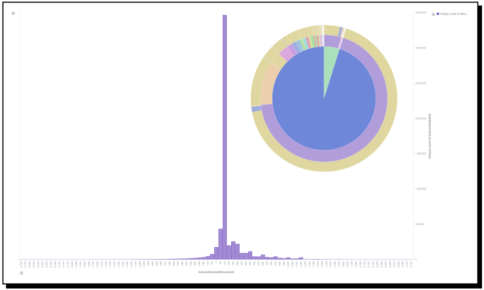
**Backup Slides:**

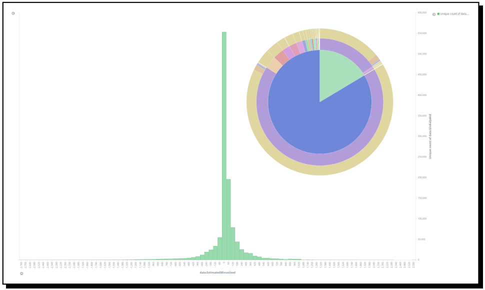
## Current implementation of Time tuning

JobCountPerTask (LogScale):



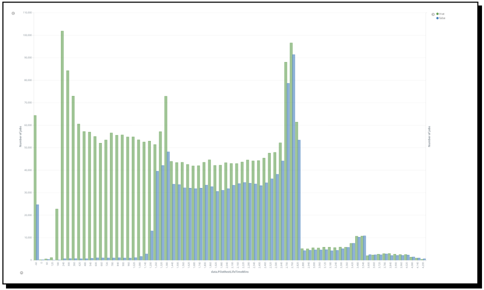
## Current results

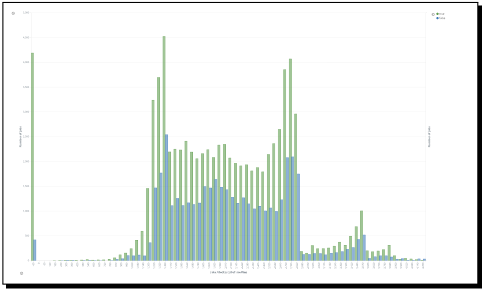


## Current results

- – Few more plots showing the results –



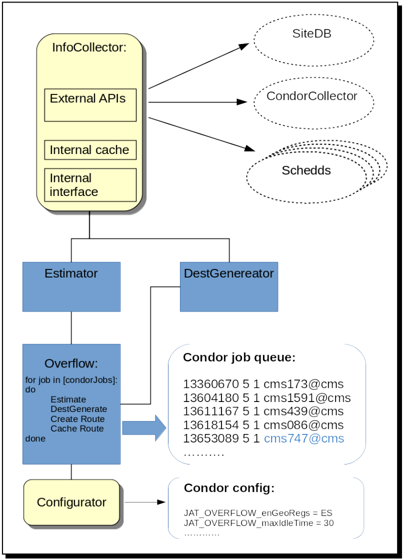
## Overflow - Problem:

- **The primary need:** to achieve a better resource utilisation
- **The secondary need:** to protect the sites from being flooded with jobs they cannot process || serve data for them.
- **The old Overflow mechanism** - what does it suffer from:
  - Statically defined overflow regions - can't be based on other criteria characterizing "proximity"
  - Overflow matching decision happens in the timescale of pilot lifetimes - not flexible enough to respond to faster changes in the status of the distributed CPU and storage resources
  - Requires additional FE groups to be set - a limitation in practice to the different number of settings that could be configured at once.
  - Based on a special type of pilots - fragmentation of the resources, increasing wastage

## Structure:

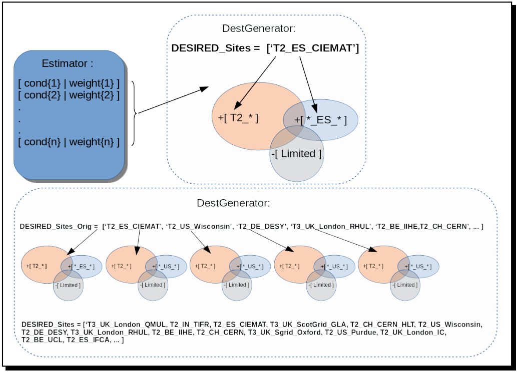
Three basic abstractions:

- **Information Lifetime:**
  - static
  - dynamic
- **The OverflowLevel:**
  - PERTASK
  - PERJOB
  - PERBLOCK
  - PERFILE
  - PERDATASET
- **The OverflowType:**
  - GEO
  - TIER1
  - TIER2
  - DATALOCATION
  - LOCALLOAD
  - SRCLOAD
  - DSTLOAD



## Decision making & Subsets

- Weighted sum vs. Weighted single decision.
- Estimating the weights could be dynamic:
  In the future we can apply more elaborate mechanisms for estimating the optimal weights according to the prompt feedback about the reaction of the system.
- Subsets intersections.



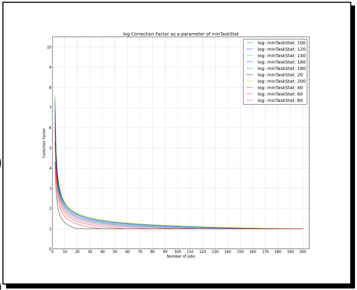
# New Methods for improving the accuracy of Automatic Time Tuning

We estimate the Job Wallclock Time (EWT) based on the first completed jobs (minTaskStat) and continuously modify the Requested Wallclock Time of the idle jobs while gaining statistics. This is a method which has the intrinsic characteristics of a negative feedback amplifier. As expected, the error with respect to the Real Time (RT) follows an normal distribution':

$$err = EWT - RT \tag{1}$$

In order to minimise this error and avoid negative values we introduce a correction factor:

$$err = CorrFactor * EWT - RT$$
$$CorrFactor = f(n)$$
$$n : numberofcompletedjobs$$




Different correction factors considered:

- static correction factor, a Heaviside function:

$$f(n) = \begin{cases} 1 & \text{if } n < \text{minTaskStat} \\ const & \text{if } n > \text{minTaskStat} \end{cases} \tag{2}$$

  minTaskStat - here is a config parameter which acts as a trigger for the mechanism

- logarithmic correction factor:

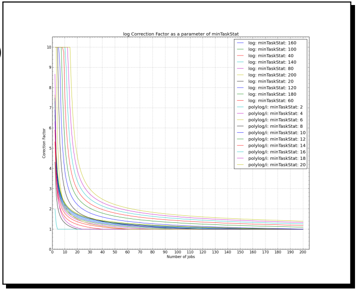
$$f(n) = \log_n(minTaskStat) \tag{3}$$

  - very steep
  - minTaskStat - is now a parameter defining the slope of the function that the correction factor will follow while gaining more statistics
  - a single parameter function
  - the negative error is still at around 16% (shows dependency on more than a single parameter)

## New Methods for improving the accuracy of Automatic Time Tuning

- polylogarithmic:
  motivation - commonly used for estimating the order of time or memory consumption

$$f(n) = \sum_{k=1}^{\epsilon} a_k (\log_n(minTaskStat))^k \tag{4}$$




– more moderate slope
– high computational cost: $O(n^{\epsilon})$ for high values of $\epsilon$
– now we can easily put more than a single parameter in the function and decide the order/degree up to which we want to calculate and $\epsilon$ becomes the number of independent parameters.
candidate parameters:

  - job dependent:
    - number of jobs in the workflow with error code diff 0
    - dataset characteristics: like number of lumisections
    - distance between slot and dataset ... etc.
  - infrastructure dependent:
    - network throughput of the slot
    - reliability of the (slot) ... etc