AUTOMAN: A Platform for Integrating Human-Based and Digital Computation

Daniel W. Barowy Charlie Curtsinger Emery D. Berger Andrew McGregor

Department of Computer Science
University of Massachusetts, Amherst
Amherst, MA 01003
{dbarowy,charlie,emery,mcgregor}@cs.umass.edu

Abstract

Humans can perform many tasks with ease that remain difficult or impossible for computers. Crowdsourcing platforms like Amazon's Mechanical Turk make it possible to harness human-based computational power at an unprecedented scale. However, their utility as a general-purpose computational platform remains limited. The lack of complete automation makes it difficult to orchestrate complex or interrelated tasks. Scheduling more human workers to reduce latency costs real money, and jobs must be monitored and rescheduled when workers fail to complete their tasks. Furthermore, it is often difficult to predict the length of time and payment that should be budgeted for a given task. Finally, the results of human-based computations are not necessarily reliable, both because human skills and accuracy vary widely, and because workers have a financial incentive to minimize their effort.

This paper introduces AUTOMAN, the first fully automatic *crowdprogramming* system. AUTOMAN integrates human-based computations into a standard programming language as ordinary function calls, which can be intermixed freely with traditional functions. This abstraction lets AUTOMAN programmers focus on their programming logic. An AUTOMAN program specifies a confidence level for the overall computation and a budget. The AUTOMAN runtime system then transparently manages all details necessary for scheduling, pricing, and quality control. AUTOMAN automatically schedules human tasks for each computation until it achieves the desired confidence level; monitors, reprices, and restarts human tasks as necessary; and maximizes parallelism across human workers while staying under budget.

OOPSLA'12, October 19–26, 2012, Tucson, Arizona, USA.

Categories and Subject Descriptors H.1.2 [*Information Systems*]: Human information processing; D.3.2 [*Language Classifications*]: Specialized application languages; G.3 [*Probability and Statistics*]: Probabilistic algorithms (including Monte Carlo)

General Terms Languages, Algorithms, Human Factors

Keywords Crowdsourcing, Programming Languages, Quality Control

1. Introduction

Humans perform many tasks with ease that remain difficult or impossible for computers. For example, humans are far better than computers at performing tasks like vision, motion planning, and natural language understanding [22, 26]. Most researchers expect these "AI-complete" tasks to remain beyond the reach of computers for the foreseeable future [27].

Recent systems streamline the process of hiring humans to perform computational tasks. The most prominent example is Amazon's Mechanical Turk, a general-purpose *crowdsourcing* platform that acts as an intermediary between labor requesters and workers [2, 15]. Domain-specific commercial services based on Mechanical Turk include CastingWords and ClariTrans, which perform accurate audio transcription, and Tagasaurus and TagCow, which perform image classification.

However, harnessing human-based computation in general and at scale faces the following challenges:

- **Determination of pay and time for tasks.** Employers must decide in advance the time allotted to a task and the payment for successful completion. It is both difficult and important to choose these correctly since workers will not accept jobs with a too-short deadline or too little pay.
- **Scheduling complexities.** Employers must manage the trade-off between latency (humans are relatively slow) and cost (more workers means more money). Because workers may fail to complete their tasks in the allotted time, jobs need to be tracked and reposted as necessary.

- **Low quality responses.** Human-based computations always need to be checked: worker skills and accuracy vary widely, and they have a financial incentive to minimize their effort. Manual checking does not scale, and simple majority voting is insufficient since workers might agree by random chance.

Contributions

This paper introduces AUTOMAN, a programming system that integrates human-based and digital computation. AUTOMAN addresses the challenges of harnessing human-based computation at scale:

Transparent integration of human and digital computation. AUTOMAN incorporates human-based computation as function calls in a standard programming language. The AUTOMAN runtime system transparently manages scheduling, budgeting, and quality control.

Automatic scheduling and budgeting. The AUTOMAN runtime system schedules tasks to maximize parallelism across human workers while staying under budget. AUTOMAN tracks job progress and reschedules and reprices failed tasks as necessary.

Automatic quality control. The AUTOMAN runtime system performs automatic quality control management. AUTOMAN automatically creates enough human tasks for each computation to achieve the confidence level specified by the programmer.

For example, given a desired confidence level of 95% and a function with five possible answers, AUTOMAN initially schedules at least three tasks (human workers). Because the chances of all three agreeing due to random chance is under 5%, a unanimous response would be considered acceptable. If all three workers do not agree, AUTOMAN will schedule another three tasks, at which point 5 out of 6 must agree to achieve a 95% confidence level (Section 5).

2. Background: Crowdsourcing Platforms

Since crowdsourcing is a novel application domain for programming language research, we start by summarizing the necessary background on crowdsourcing platforms. Our discussion in this section focuses on Amazon's Mechanical Turk, but other existing crowdsourcing platforms are similar.

Mechanical Turk acts as an intermediary between employers (known as *requesters*) and employees (*workers*, or colloquially, *turkers*) for short-term assignments.

Human Intelligence Tasks (HITs). In Mechanical Turk parlance, individual tasks are known as *HITs*, which stands for *human intelligence tasks*. HITs include a short description, the amount the job pays, and other details. Most HITs on Mechanical Turk are for relatively simple tasks, such as "does this image match this product?" Compensation is generally low, since employers expect that work can be completed on a time scale ranging from seconds to minutes. Pay for HITs range from a single penny to several dollars.

Each HIT is represented as a question form, composed of any number of questions, and associated metadata such as a title, description, and search keywords. Questions can be one of two types: a free text question, where workers can provide a free-form textual response, or a multiple-choice question, where workers make one or more selections from a list of possible options. We refer to the former as *open-ended questions* and the latter as *closed-ended questions*; AUTOMAN currently only supports closed-ended questions.

Requesters: Posting HITs. Mechanical Turk allows HITs to be posted manually, but also exposes a web service API that allows basic details of HITs to be managed programmatically [2], including posting HITs, collecting completed work, and paying workers. Using this API, it is straightforward to post similar tasks to Mechanical Turk *en masse*. HITs sharing similar qualities can be grouped into *HIT groups*.

A requester may also instruct Mechanical Turk to parallelize a particular HIT by indicating whether each HIT should be assigned to more than one worker. By increasing the number of assignments, Mechanical Turk allows additional workers to accept work for the same HIT, and the system ensures that the parallel workers are unique (i.e., that a single worker cannot complete the same HIT more than once).

Workers: Performing Work. Mechanical Turk workers can choose any of the available tasks on the system for which they are qualified (see below): as of this writing, there are approximately 275,000 HITs posted. When workers choose to perform a particular HIT, they accept an *assignment*, which grants them a time-limited reservation for that particular piece of work; that is, no other worker may accept it.

HIT Expiration. HITs have two timeout parameters: the amount of time that a particular HIT should remain in the listings, known as the *lifetime* of a HIT, and the amount of time that a worker has to complete an assignment once it is accepted, known as the *duration* of an assignment. If after accepting an assignment for a HIT, a worker exceeds the assignment's duration without submitting completed work, the reservation is cancelled, and the work is returned to the pool of available assignments. If a HIT reaches the end of its lifetime without its assignments having been completed, the HIT expires and is removed from the job board.

Requesters: Accepting or Rejecting Work. Once a worker completes and submits an assignment, the requester is notified. The requester then can accept or reject the completed assignment. Acceptance of an assignment indicates that the completed work is satisfactory, and the worker is then automatically paid for his or her efforts. Rejection withholds payment, and the requester, if so inclined, may provide a textual justification for the rejection. AUTOMAN automatically manages acceptance and rejection; see Section 3.2.

Worker Quality Control. A key challenge in automating work in Mechanical Turk is attracting and retaining good workers, or at least discouraging bad workers from participating. However, Mechanical Turk provides no way for requesters to seek out specific workers.

Instead, Mechanical Turk provides a *qualification* mechanism to limit which workers may perform a particular HIT. A common qualification is that workers must have an overall assignment-acceptance rate of 90%. However, given the wide variation in tasks on Mechanical Turk, overall worker accuracy is of limited utility.

For example, the fact that a worker who is skilled in and favors audio transcription tasks may have a high accuracy rating, but there is no reason to believe that this worker can also perform Chinese-to-English language translation tasks. Worse, workers who cherry-pick easy tasks and thus have a high accuracy rating actually may be less qualified than a worker who routinely performs difficult work that is occasionally rejected.

3. Overview

AUTOMAN is a domain-specific language embedded in Scala [24]. AUTOMAN's goal is to abstract away the details of crowdsourcing so that human computation can be as easy to invoke as a conventional function.

3.1 Using AUTOMAN

Figure 1 presents an example (toy) AUTOMAN program. The program "computes" which of a set of cartoon characters does not belong in the group. Notice that the programmer does not specify details about the chosen crowdsourcing backend (Mechanical Turk) except for account credentials.

Crucially, all details of crowdsourcing are hidden from the AUTOMAN programmer. The AUTOMAN runtime manages interfacing with the crowdsourcing platform, schedules and determines budgets (both cost and time), and automatically ensures the desired confidence level of the final result.

***Initializing* AUTOMAN.** After importing the AUTOMAN and Mechanical Turk adapter libraries, the first thing an AUTOMAN programmer does is to declare a configuration for the desired crowdsourcing platform. This configuration is then bound to an AUTOMAN runtime object, which instantiates any platform-specific objects.

Specifying* AUTOMAN *functions. Functions in AUTOMAN consist of declarative descriptions of questions that the workers must answer; they may include text or images, as well as a range of question types, which we describe below.

Confidence level. An AUTOMAN programmer can optionally specify the degree of confidence they want to have in their computation, on a per-function basis. AUTOMAN's default confidence is 95% (0.95), but this can be overridden as needed. The meaning and derivation of confidence is discussed in Section 5.

```
1  import edu.umass.cs.automan.adapters.MTurk._
2
3  object SimpleProgram extends App {
4    val a = MTurkAdapter { mt =>
5      mt.access_key_id = "XXXX"
6      mt.secret_access_key = "XXXX"
7    }
8
9    def which_one() = a.RadioButtonQuestion { q =>
10     q.budget = 8.00
11     q.text = "Which one of these does not belong?"
12     q.options = List(
13       a.Option('oscar, "Oscar the Grouch"),
14       a.Option('kermit, "Kermit the Frog"),
15       a.Option('spongebob, "Spongebob Squarepants"),
16       a.Option('cookie, "Cookie Monster"),
17       a.Option('count, "The Count")
18     )
19   }
20
21   println("The answer is " + which_one()())
22 }
```

Figure 1. A *complete* AUTOMAN program. This program computes, by invoking humans, which cartoon character does not belong in a given set. The AUTOMAN programmer specifies only credentials for Mechanical Turk, an overall budget, and the question itself; the AUTOMAN runtime manages all other details of execution (scheduling, budgeting, and quality control).

Metadata and question text. Each question requires a title and description, used by the crowdsourcing platform's user interface. These fields map to Mechanical Turk's fields of the same name. A question also includes a textual representation of the question, together with a map between symbolic constants and strings for possible answers.

Question variants. AUTOMAN supports multiple-choice questions, including questions where only one answer is correct ("radio-button" questions), or where any number of answers may be correct ("checkbox" questions), as well as restricted-text entry forms. Section 5 describes how AUTOMAN's quality control algorithm handles these different types of questions.

Invoking a function. An AUTOMAN programmer can invoke a function as if it were any ordinary (digital) function. Here, the programmer calls the just-defined function `which_one()` with no input. The function returns a Scala future object representing the answer, which can be passed to other `Question`s in an AUTOMAN program before the human computation is complete. AUTOMAN functions execute in the background in parallel as soon as they are invoked. The program does not block until it references the function output, and only then if the human computation is not yet finished.

3.2 AUTOMAN Execution

Figure 2 depicts an actual trace of the execution of the program from Figure 1, obtained by executing it with Amazon's Mechanical Turk. This example demonstrates that ensuring valid results even for simple programs can be complicated.

Starting Tasks. At startup, AUTOMAN examines the form of the question field defined for the task and determines that, in order to achieve a 95% confidence level for a question with five possible choices, at minimum, it needs three different workers to unanimously agree on the answer (see Section 5). AUTOMAN then spawns three tasks on the crowdsourcing backend, Mechanical Turk. To eliminate bias caused by the position of choices, AUTOMAN randomly shuffles the order of choices in each task.

AUTOMAN's default strategy is optimistic. For many tasks, human workers are likely to agree unanimously. Whenever this is true, AUTOMAN saves money by spending the least amount required to achieve the desired statistical confidence.

However, AUTOMAN also allows users to choose a more aggressive strategy that trades a risk of increased cost for reduced latency; see Section 4.3.

Quality Control. At time 1:50, worker 1 accepts the task and submits "Spongebob Squarepants" as the answer. Forty seconds later, worker 2 accepts the task and submits the same answer. However, twenty seconds later, worker 3 accepts the task and submits "Kermit". In this case, AUTOMAN's optimism did not pay off, since worker 3 does not agree with worker 1 and 2. Because this result is inconclusive, AUTOMAN schedules three additional tasks.

At this point, AUTOMAN recomputes the minimal number of agreements, which turns out to be five out of six. Section 5.2 presents a full derivation of the closed-form formulas that AUTOMAN uses to compute these values.

Memoization of Results. During program execution, AUTOMAN saves the intermediate state of all human functions, namely scheduler objects, to a database. This means that if a program is interrupted, AUTOMAN's scheduler is able to resume the program precisely where it left off. Memoization is automatic and transparent to the programmer. This abstraction serves two purposes. First, the AUTOMAN function may be called an arbitrary number of times, resulting in substantial time savings. Second, if the program's execution is interrupted in any way, the programmer may resume the program without losing their investment in human labor. Programmers who do not want to use memoized results need only delete the memo database at startup.

Initial Time Estimates. When defining a task, the programmer can specify the task "duration", the time available to work once the task has been accepted. On Mechanical Turk, this number serves as an indication to the worker of the difficulty of the task. In AUTOMAN, this figure is set to 30 seconds by default.

A second time parameter, the "lifetime" of a task, indicates how long the crowdsourcing backend should keep the task around without any response from workers. In AUTOMAN, the task lifetime is set to 100× the duration.

At 51 minutes into the computation, task 6 exceeds its lifetime and is cancelled. Since AUTOMAN does not yet have

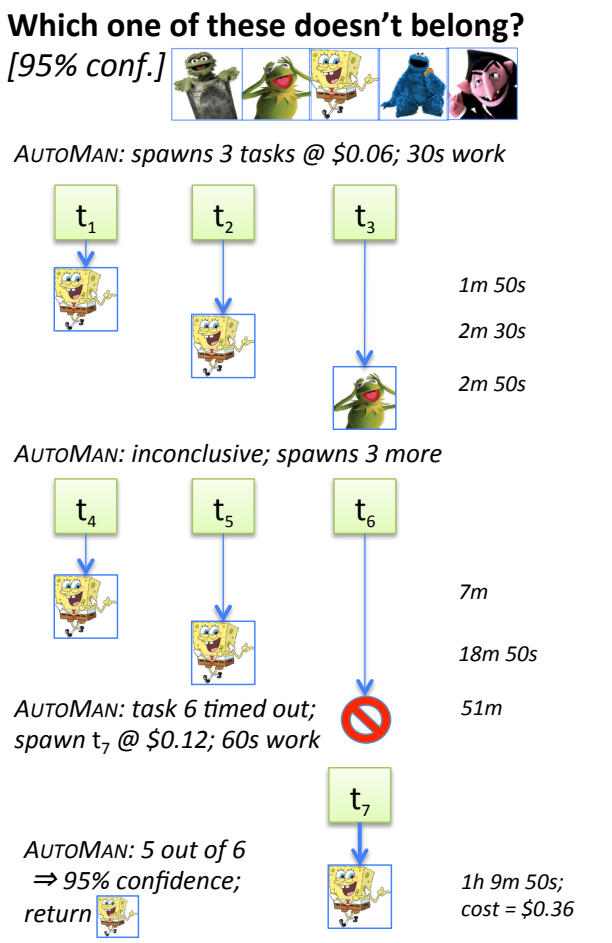


Figure 2. A schematic of an actual execution of the example program in Figure 1, with time advancing from top to bottom. AUTOMAN first spawns 3 tasks, which (for five choices) suffice to reach a 95% confidence level if all workers agree. Since the 3 workers do not agree, AUTOMAN schedules three additional tasks (5 of 6 must now agree). When task 6 times out, AUTOMAN spawns a new task and doubles both the budget and time allotted. Once task 7 completes, AUTOMAN returns the result.

a statistically valid result, it reschedules the task, this time extending both the task timeout and lifetime by a factor of two (see Section 4.1).

Rebudgeting: Time and Pay. AUTOMAN does not require the programmer to specify exactly how much each worker should be paid. AUTOMAN currently uses the timeout parameter and calculates the base cost for the task using the US Federal minimum wage ($7.25/hr). For 30 seconds of work, the payment is $0.06 US Dollars.

AUTOMAN automatically manages task rewards and durations by doubling both whenever a task's lifetime expires, which maintains the same minimum wage (see Section 4.1). In the absence of an automatic mechanism, programmers would be required to determine the fair wage of the task mar-

ketplace manually. Given the subjectivity of a "fair wage," knowing the appropriate wage *a priori* is difficult or impossible in most cases.

Automatic Task Acceptance and Rejection. AUTOMAN does not accept or reject work until the quality control algorithm has chosen the correct answer. Payment is deferred until the computation completes. If the programmer's account runs out of funds before the program is complete, AUTOMAN returns the answer with the highest confidence. The programmer may then choose to either add more money to their account and resume the computation, using stored results from the memoization database, or to accept the lower-quality answer and pay all of the workers.

One hour and 9 minutes into the computation, a worker submits a sixth answer, "Spongebob Squarepants". AUTOMAN again examines whether the answers agree and it finds that 5 out of the 6 answers agree. AUTOMAN can now reject the null hypothesis, i.e., that 5 workers agreed by choosing the same answer randomly, with a 95% confidence level. The runtime system then returns the answer to the program, and the user's regular program resumes.

AUTOMAN then informs the crowdsourcing backend to pay the five workers whose answers agreed. Four workers are paid $0.06, and one is paid $0.12. The one worker whose answer did not agree was not paid. For Mechanical Turk, which supports rejection notifications, AUTOMAN informs workers who provided incorrect responses that their work was not accepted, and includes the correct response and confidence as justification.

The fact that AUTOMAN does not pay for incorrect work reduces its cost, especially as the number of workers increases. For example, if 12 responses have been received for a question with 5 choices, only 7 must agree to achieve a 95% confidence level. As the number of workers increases, the proportion required for agreement drops further, making rejecting incorrect work even more desirable.

```
1  wage = DEFAULT_WAGE
2  value_of_time = DEFAULT_VALUE_OF_TIME
3  duration = DEFAULT_DURATION
4  reward = wage * duration
5  budget = DEFAULT_BUDGET
6  cost = $0.00
7  tasks = []
8  answers = load_saved_answers()
9  timed_out = false
10 confident = false
11
12 while not confident:
13   if timed_out:
14     duration *= 2
15     reward *= 2
16     timed_out = false
17
18   if tasks.where(state == RUNNING).size == 0:
19     most_votes = answers.group_by(answer).max
20     required = 0
21     while min_votes(choices, most_votes + required)
22         > most_votes + required:
23       required += 1
24
25     if required == 0:
26       confident = true
27     else
28       can_afford = floor((budget - cost) / reward)
29       if can_afford < required:
30         throw OVER_BUDGET
31       ideal = floor(value_of_time / wage)
32       to_run = max(required, min(can_afford, ideal))
33       cost += to_run * reward
34       tasks.appendAll(spawn_tasks(to_run))
35
36   else:
37     num_timed_out = tasks.where(state == TIMEOUT).size
38     if num_timed_out > 0:
39       timed_out = true
40       cost -= num_timed_out * reward
41     foreach t in tasks.where(state == ANSWERED):
42       answers.append(t.answer)
43       save_answer(t.answer)
44 return answers.group_by(answer).argmax
```

Figure 3. Pseudo-code for AUTOMAN's scheduling loop, which handles posting and re-posting jobs, budgeting, and quality control; Section 5.2 includes a derivation of the formulas for the quality control thresholds.

4. Scheduling Algorithm

Figure 3 presents pseudo-code for AUTOMAN's main scheduler loop, which comprises the algorithms that the AUTOMAN runtime uses to manage task posting, reward and timeout calculation, and quality control.

4.1 Calculating Timeout and Reward

AUTOMAN's overriding goal is to recruit workers quickly and at low cost in order to keep the cost of a computation within the programmer's budget. AUTOMAN posts tasks in rounds, which have a fixed timeout during which tasks must be completed. When AUTOMAN fails to recruit workers in a round, there are two possible causes: workers were not willing to complete the task for the given reward, or the time allotted was not sufficient. AUTOMAN does not distinguish between these cases. Instead, the reward for a task and the time allotted are both increased by a constant factor k every time a task goes unanswered. k must be chosen carefully to ensure the following two properties:

1. The reward for a task should quickly reach a worker's minimum acceptable compensation (R_{min}), e.g., in a logarithmic number of steps.
2. The reward should not grow so quickly that it would give workers an incentive to wait for a larger reward, rather than work immediately.

Workers do not know the probability that a task will remain unanswered until the next round. If the worker assumes even odds that a task will survive the round, a growth rate of $k = 2$ is optimal: it will reach R_{min} faster than any lower value of k, and workers never have an incentive to wait. Section 4.4 presents a detailed analysis. Lines 13-16 in Figure 3 increase the reward and duration for tasks that have timed out.

In AUTOMAN, reward and time are specified in terms of the worker's wage ($7.25/hour for all the experiments in this paper). Doubling both reward and time ensures that AUTOMAN will never exceed the minimum time and reward by more than a factor of two.

The doubling strategy may appear to run the risk that a worker will "game" the computation into paying a large sum of money for an otherwise simple task. However, once the wage reaches an acceptable level for some proportion of the worker marketplace, those workers will accept the task. Forcing AUTOMAN to continue doubling to a very high wage would require collusion between workers on a scale that we believe is infeasible, especially when the underlying crowdsourcing system provides strong guarantees that worker identities are independent.

4.2 Scheduling the Right Number of Tasks

AUTOMAN's default strategy for spawning tasks is optimistic: it creates the smallest number of tasks required to reach the desired confidence level if the results are unanimous. Line 19 in Figure 3 determines the number of votes for the most popular answer so far. Lines 20-23 iteratively compute the minimum number of additional votes required to reach confidence. If no additional votes are required, confidence has been reached and AUTOMAN can return the most popular answer (line 44).

Using the current reward, AUTOMAN computes the maximum number of tasks that can be posted with the remaining budget (line 28). If the budget is insufficient, AUTOMAN will terminate the computation, leaving all tasks in an unverified state (lines 29-30). The computation can be resumed with an increased budget or abandoned. Mechanical Turk will automatically pay all workers if responses are not accepted or rejected after 30 days.

4.3 Trading Off Latency and Money

AUTOMAN also allows programmers to provide a *time-value* for the computation, which tells AUTOMAN to post more than the minimum number of tasks. AUTOMAN always schedules at least the minimum number of tasks required to achieve confidence in every round. If the programmer's time is worth more than the total cost of the minimum number of tasks, $\left\lfloor \frac{\text{time\_value}}{\text{min\_wage}} \right\rfloor$ tasks will be scheduled instead (lines 31-32). Once AUTOMAN receives enough answers to reach the specified confidence, it cancels any outstanding tasks. In the worst case, all posted tasks will be answered before AUTOMAN can cancel them, which will cost no more than time_value $\cdot$ task_timeout.

This strategy runs the risk of paying substantially more for a computation, but can yield dramatic reductions in latency. We re-ran the example program given in Figure 1 with a time-value set to $50, 7× larger than the current U.S. minimum wage. In two separate runs, the computation completed in 68 and 168 *seconds*; we also ran the first computation with the default time-value (minimum wage), and those computations took between 1 and 3 *hours* to complete.

4.4 Derivation of Optimal Reward Growth Rate

When workers encounter a task with a posted reward of R, they may choose to accept the task, or wait for the reward to grow. Let p_a be the probability that the task will still be available after one round of waiting. We make the assumption that, if the task is still available after $i-1$ rounds, then the probability that the task is available in the ith round is at most p_a. Hence, if the player's strategy is to wait i rounds and then complete the task,

$$\mathbb{E}\left[\text{reward}\right] \leq p_a^i k^i R\,,$$

since with probability at most p_a^i the reward will be $k^i R$ and otherwise the task will no longer be available.

Note that the expected reward is maximized with $i = 0$ if $k \leq 1/p_a$. Therefore, $k = 1/p_a$ is the highest value of k that does not incentivize waiting and will reach R_{min} faster than any lower value of k. Workers cannot know the true value of p_a. In the absence of any information, $1/2$ will be used as an estimator for p_a and this leads to AUTOMAN's default value of $k = 2$.

However, it is possible to estimate p_a. Every time a worker accepts or waits for a task, we can treat this as an independent Bernoulli trial with the parameter p_a. The maximum likelihood estimator for p_a equals

$$\tilde{p}_a = \operatorname*{argmax}_{x \in [0,1]} x^t (1-x)^{n-t}$$

where t is the number of times a task has been accepted amongst the n times it has been offered so far. Solving this gives $\tilde{p}_a = t/n$.

The difficulty of accurately estimating p_a using *ad hoc* quality control is a strong argument for automatic budgeting. Implementing this estimation is a planned future enhancement for AUTOMAN.

5. Quality Control Algorithm

AUTOMAN's quality control algorithm is based on collecting enough consensus for a given question to rule out the possibility, with a desired level of confidence, that the results are due to random chance. Section 5.3 justifies this approach.

Initially, AUTOMAN spawns enough tasks to meet the desired confidence level if all workers who complete the tasks agree on the same answer. Figure 5 depicts the initial confidence level function. Computing this value is straightforward: if k is the number of choices, and n is the number of tasks, the confidence level reached is $1 - k(1/k)^n$. AUTOMAN computes the lowest value of n where the desired confidence level is reached.

Question Variants. For ordinary multiple choice questions where only one choice is possible ("radio-button" questions),

k is exactly the number of possible answers. However, humans are capable of answering a richer variety of question types. Each of these additional question types requires its own probability analysis.

Checkbox Questions. For multiple choice questions with c choices and any or all may be chosen ("checkbox" questions), k is much larger: $k = 2^c$.

For these questions, k is so high that a very small number of workers are required to reject the null hypothesis (random choice). However, it is reasonably likely that two lazy workers will simply select no answers, and AUTOMAN will erroneously accept that answer is correct.

To compensate for this possibility, AUTOMAN treats checkbox questions specially. The AUTOMAN programmer must specify not only the question text, but also an *inverted* question. For instance, if the question is "Select which of these are true", the inverted question should read, "Select which of these are NOT true." AUTOMAN then ensures that half of the HITs are spawned with the positive question, and half with the inverted question. This strategy makes it less likely that lazy workers will inadvertently agree.

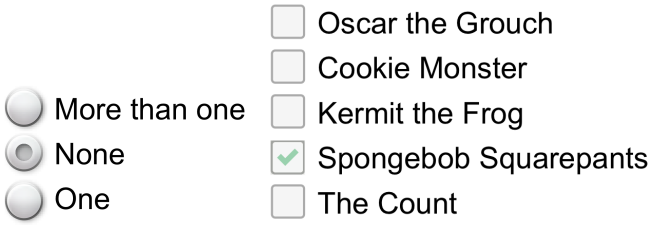
Restricted Free-Text Questions Entering a small amount of text into a text input is essentially equivalent to a sequence of radio-button questions, where the possible radio-button options are the valid range of textual inputs for the given question. This form of input is cumbersome for human workers. By providing a small text field with a pattern representing valid inputs, AUTOMAN satisfies its requirement for analysis while making data entry easier.

The input specification in AUTOMAN resembles COBOL's picture clauses. For example, a telephone number recognition application would use the pattern `09999999999`. `A` matches an alphabetic character, `B` matches an optional alphabetic character, `X` matches an alphanumeric character, `Y` matches an optional alphanumeric character, `9` matches a numeric character, and `0` matches an optional numeric character. All other characters only match themselves (e.g., "-" matches "-"). This particular pattern will match a string with nine or ten numeric characters.

Assuming a uniform probability for each possibility, calculating the number of matching 7-character numeric strings is straightforward: $k = 10^7$. Again, k is often very large, so a small number of HITs suffice to achieve high confidence in the result.

As with checkbox questions, AUTOMAN must treat free-text questions specially to cope with "lazy" workers. Here a lazy worker might simply hit the "Submit" button, submitting the empty string and biasing the answer. To avoid this problem, AUTOMAN only accepts the empty string if it is explicitly allowed as a legal input. In this case, AUTOMAN requires that workers enter a special value, `N/A`.

We are currently investigating the use of a subset of regular expressions as input specifications. Regular expressions are far more powerful than COBOL picture clauses, but also more difficult to analyze. The number of matching strings must be bounded, otherwise the probability calculation is not possible. This limitation rules out the use of the `*`, `+`, and `{x,}` (where `x` is an integer) operators in patterns. Alternation must also be handled carefully as the pattern `a?a?b` matches only three strings, `b`, `ab`, and `aab`. A naïve counting implementation would count `ab` twice.



(a) A radio-button question. $k = c$.

(b) A checkbox question. $k = 2^c$.

(c) A free-text question. $k = \prod_{i=0}^{l} p_i$ where l is the length of the pattern and p_i is the number of distinct characters matched by the picture clause at position i.

Figure 4. Question types handled by AUTOMAN.

5.1 Overview of the Quality Control Algorithm

In order to return results within the user-defined confidence interval, AUTOMAN iteratively calculates two threshold functions that tell it whether it has enough confidence to terminate, and if not, how many additional workers it must recruit. Formally, the quality control algorithm depends on two functions, t and ℓ, and associated parameters α, β, and p^*. The $t(n, \alpha)$ and $\ell(p^*, \beta)$ functions are defined such that:

- $t(m, \alpha)$ is the threshold number of agreeing votes. If the workers vote randomly (i.e., each answer is chosen with equal probability), then the probability that an answer meets the threshold of $t(n, \alpha)$ when n votes are cast is at most α. α will be determined based on the confidence parameter chosen by the programmer (α = 1 - confidence).
- $\ell(p^*, \beta)$ is the minimum number of additional workers to recruit for the next step. If there is a "popular" option such that the probability a worker chooses it is p and $p > p^*$ (and all other options are equally likely), then if AUTOMAN receives votes from $n \geq \ell(p^*, \beta)$ workers some answer will meet the threshold $t(n, \alpha)$ with probability at least $1 - \beta$.

We will derive expressions for t and ℓ in Section 5.2. The algorithm proceeds as follows:

1. Set $n = \min\{m : t(m, \alpha) \neq \infty\}$

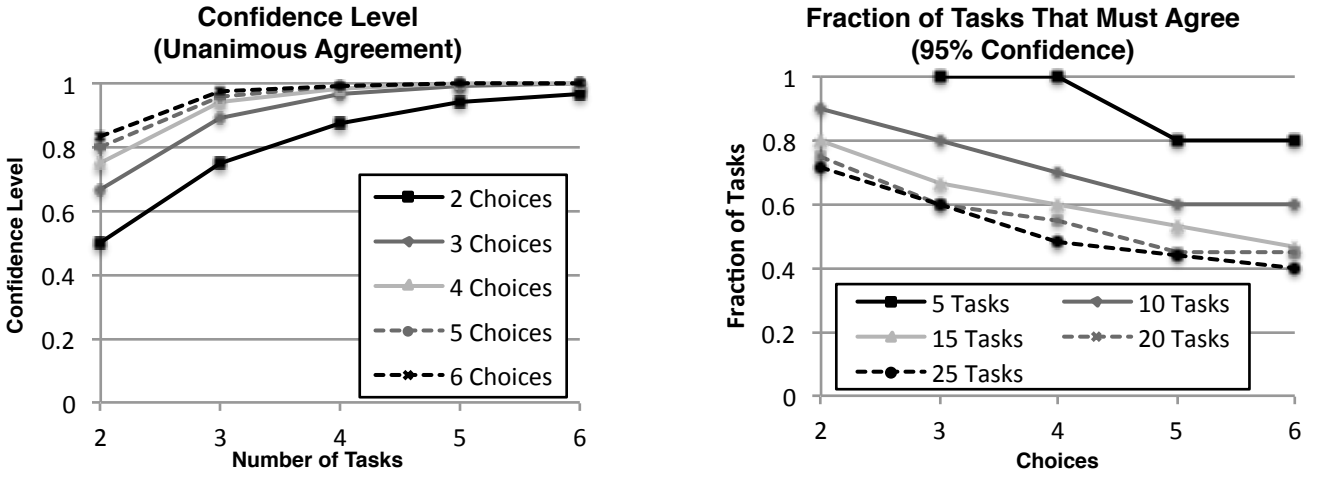


Figure 5. The confidence level reached with unanimous agreement (left) and fraction of tasks (workers) that must agree to reach 95% confidence. As the the number of choices increases, the number of tasks required to reach confidence quickly drops. Similarly, the fraction of tasks that must agree drops as choices and tasks are added.

2. Ask n workers to vote on the answer of a question given k options.
3. If there are options that have more than $t(n, \alpha)$ votes, return the most frequent option of all of the options that are above the threshold.
4. If $n < \ell(p^*, \beta)$, double n and repeat from step 2.

Figure 5 uses the value of t to compute the normalized *fraction* of tasks that need to agree in order to reach $\alpha = 0.05$ (a confidence level of 95%). As the number of tasks and the number of choices increase, the fraction of the number of tasks needed for agreement decreases. For example, with 25 tasks and a question with 4 choices, only 48% (12 of 25) need to agree in order to achieve 95% confidence. In other words, as the number of workers increases, AUTOMAN needs an ever smaller fraction of those workers to agree, speeding convergence to a correct answer.

5.2 Derivation of $\ell(p^*, \beta)$ and $t(n, \alpha)$

Given parameters $n \in \mathbb{N}_0$ and $0 \leq p_1, p_2, \ldots, p_k \leq 1$ with $\sum_{i=1}^{k} p_i = 1$, $Z = (Z_1, \ldots, Z_k)$ is a multinomial distribution with parameters $(n, p_1, p_2, \ldots, p_k)$ if for any $z_1, z_2, \ldots \in \mathbb{N}_0$ with $\sum_{i=1}^{k} z_i = n$

$$\Pr[\forall i : \ Z_i = z_i] = \frac{n!}{z_1! z_2! \ldots z_k!} p_1^{z_1} \ldots p_k^{z_k}$$

For example, if n voters are given k options and each voter (independently) picks option i with probability p_i then Z_i will correspond to the number of votes received by the ith option.

To compute the probability of a multinomial distribution, we follow the approach outlined by DasGupta [7].

Lemma 5.1. *For* $0 \leq a_i \leq b_i \leq n$ *and* $i \in [k]$,

$$\Pr[\forall i : \ Z_i \in [a_i, b_i]] = n! \cdot \text{coeff}_{\lambda,n} \prod_{i=1}^{k} \sum_{j=a_i}^{b_i} \frac{(p_i \lambda)^j}{j!} ,$$

where $\text{coeff}_{\lambda,n}(f(\lambda))$ *is the coefficient of* λ^n *in the polynomial* f.

Proof. Let $N \in \text{Poi}(\lambda)$ be a random variable distributed according to the Poisson distribution with parameter λ. Then let $Z^N = (Z_1^N, \ldots, Z_k^N)$ is a multinomial distribution with parameters $(N, p_1, p_2, \ldots, p_k)$. Note that $Z = Z^n$. It is known (e.g., [12, pp. 216]) that

$$\begin{aligned} \Pr[\forall i : \ Z_i^N \in [a_i, b_i]] &= \prod_{i=1}^{k} \Pr[Z_i^N \in [a_i, b_i]] \\ &= \prod_{i=1}^{k} \sum_{j=a_i}^{b_i} \frac{(p_i \lambda)^j e^{-p_i \lambda}}{j!} \\ &= e^{-\lambda} \prod_{i=1}^{k} \sum_{j=a_i}^{b_i} \frac{(p_i \lambda)^j}{j!} . \end{aligned}$$

But

$$\begin{aligned} &\Pr[\forall i : \ Z_i^N \in [a_i, b_i]] \\ = &\sum_{m=0}^{\infty} \Pr[N = m] \cdot \Pr[\forall i : \ Z_i^m \in [a_i, b_i]] \\ = &\sum_{m=0}^{\infty} \frac{\lambda^m e^{-\lambda}}{m!} \cdot \Pr[\forall i : \ Z_i^m \in [a_i, b_i]] \end{aligned}$$

and hence

$$\sum_{m=0}^{\infty} \frac{\lambda^m}{m!} \cdot \Pr[\forall i : \ Z_i^m \in [a_i, b_i]] = \prod_{i=1}^{k} \sum_{j=a_i}^{b_i} \frac{(p_i \lambda)^j}{j!} .$$

The result follows by equating the coefficients of λ^n. $\square$

Let X and Y be multinomial distributions with parameters $(n, 1/k, \ldots, 1/k)$ and $(n, p, q, \ldots, q)$ where $q = (1-p)/(k-1)$ respectively. The following corollary follows immediately from Lemma 5.1.

Corollary 5.1.

$$\begin{aligned} \Pr\left[\max_i X_i < t\right] &= \mathcal{E}_1(n,t) \\ \Pr\left[\max_{i\geq 2} Y_i < t \leq Y_1\right] &= \mathcal{E}_2(p,n,t) \end{aligned}$$

where

$$\mathcal{E}_1(n,t) = \frac{n!}{k^n} \cdot \mathrm{coeff}_{\lambda,n} \left(\sum_{j=0}^{t-1} \lambda^j / j! \right)^k$$

and

$$\mathcal{E}_2(p,n,t) = n! \cdot \mathrm{coeff}_{\lambda,n} \left(\sum_{j=0}^{t-1} \frac{(q\lambda)^j}{j!} \right)^{k-1} \cdot \sum_{j=t}^{\infty} \frac{(p\lambda)^j}{j!} \,.$$

Note that $\mathcal{E}_1(n,n) = 1 - 1/k^{n-1}$ and define

$$t(n,\alpha) := \begin{cases} \min\{t : \mathcal{E}_1(n,t) \geq \delta\} & \text{if } \mathcal{E}_1(n,n) \geq \delta \\ \infty & \text{if } \mathcal{E}_1(n,n) < \delta \end{cases}.$$

where $\delta = 1 - \alpha$. This ensures that when n voters each randomly chose an option, the probability that an option meets or exceeds the threshold $t(n,\alpha)$ is at most α.

Next we define

$$\ell(p^*, \beta) := \min\{n : \mathcal{E}_2(p^*, n, t(n,\alpha)) \geq 1 - \beta\} \,.$$

If the voters have a bias of at least p^* towards a certain popular option, and all other options are equally weighted, then by requiring $\ell(p^*, \beta)$ voters, AUTOMAN ensures that the number of votes cast for the popular option crosses the threshold (and all other options are below threshold) with probability at least $1 - \beta$.

5.3 Quality Control Discussion

For AUTOMAN's quality control algorithm to work, two assumptions must hold:

- Workers are independent.
- Random choice is the worst-case behavior for workers; that is, they will not deliberately pick the wrong answer.

Workers may break the assumption of independence in three ways: (1) a single worker may masquerade as multiple workers; (2) a worker may perform multiple tasks; and (3) workers may collude when working on a task.

Scenario 1: Sybil Attack. The first scenario, where one real user creates multiple electronic identities, is known in the security literature as a "Sybil attack" [9]. The practicality of a Sybil attack depends directly on how easy it is to generate multiple identities.

Carrying out a Sybil attack on our default backend, Mechanical Turk, would be extremely burdensome. Since Mechanical Turk provides a payment mechanism for workers, Amazon requires that workers provide uniquely identifying financial information, typically a credit card or bank account. These credentials are difficult, although not impossible, to forge.

Scenario 2: One Worker, Multiple Tasks. AUTOMAN avoids the second case (one worker performing multiple tasks) via a workaround of Mechanical Turk's existing mechanisms. Mechanical Turk provides a mechanism to ensure worker uniqueness for a given HIT (*i.e.*, a HIT with multiple assignments), but it lacks the functionality to ensure worker uniqueness across multiple HITs. For example, when AUTOMAN decides to respawn a task, it must be certain that workers who participated in previous instantiations of that task are excluded from future instantiations.

Our workaround for this shortcoming is to use Mechanical Turk's "qualification" feature in an inverse sense. Once a worker completes a HIT that is a part of a larger computation, AUTOMAN grants that worker special qualification (effectively, a "disqualification") that *precludes* them from participating in future tasks of the same kind. Our system ensures that workers are not able to request reauthorization.

Scenario 3: Worker Collusion. While it would be possible to attempt to lower the risk of worker collusion by ensuring that they are geographically separate (e.g., by filtering workers using IP geolocation), AUTOMAN currently does not take any particular action to prevent worker collusion. Preventing this scenario is essentially impossible. Nothing prevents workers from colluding via external channels (e-mail, phone, word-of-mouth) to thwart the assumption of independence. Instead, the system should be designed to make the effort of thwarting defenses undesirable given the payout.

By spawning large numbers of tasks, AUTOMAN makes it difficult for any single group to monopolize them. In Mechanical Turk, no one worker has a global view of the system, thus the state of AUTOMAN's scheduler is unknown to the worker. Without this information, workers cannot game the system. The prevalent behavior is that people try to do as little work as possible to get compensated: previous studies of Mechanical Turk indicate random-answer spammers are the primary threat. [28].

5.3.1 Random as Worst Case

AUTOMAN's quality control function is based on excluding the possibility of random choices by workers; that is, workers who minimize their effort or make errors. It is possible that workers could instead act maliciously and deliberately choose

incorrect answers. Participants in crowdsourcing systems have both short-term and long-term economic incentives to not deliberately choose incorrect answers, and thus random choice is a reasonable worst-case scenario.

First, a correct response to a given task yields an immediate monetary reward. If a worker has any information about what the correct answer is, it is against their own short-term economic self-interest to deliberately avoid it. In fact, as long as there is a substantial bias towards the correct answer, AUTOMAN's algorithm will eventually accept it.

Second, while a participant might out of malice choose to forego the immediate economic reward, there are long-term implications for deliberately choosing incorrect answers. Crowdsourcing systems like Mechanical Turk maintain an overall ratio of accepted answers to total answers submitted, and many requesters place high qualification bars on these ratios (typically around 90%). Incorrect answers thus have a lasting negative impact on workers, who, as mentioned earlier, cannot easily discard their identity and adopt a new one.

Anecdotal evidence from our experience supports these assumptions. Mechanical Turk workers have contacted us when AUTOMAN rejects their answers (AUTOMAN provides the correct answer in its rejection notice). Many workers sent us e-mails justifying their answers or apologizing for having misunderstood the question, requesting approval to maintain their overall ratio of correct to incorrect responses. We typically approved payment for workers who justified their incorrect answers, but Mechanical Turk does not allow us to accept already-rejected HITs.

6. System Architecture and Implementation

In order to cleanly separate the concerns of delivering reliable data to the end-user, interfacing with an arbitrary crowdsourcing system, and specifying validation strategies in a crowdsourcing system-agnostic manner, AUTOMAN is implemented in tiers.

6.1 Domain-specific language

The programmer's interface to AUTOMAN is a set of function calls, implemented as an embedded domain-specific language for the Scala programming language. The choice of Scala as a host language was motivated primarily by the desire to have access to a rich set of language features while maintaining compatibility with existing code. Scala is fully interoperable with existing Java code; the crowdsourcing system compatibility layer heavily utilizes this feature to communicate with Amazon's Mechanical Turk system. Scala also provides access to powerful functional language features that simplify the task of implementing a complicated system. These function calls act as syntactic sugar, strengthening the illusion that crowdsourcing tasks really are just a kind of function call with an extra error tolerance parameter. Scala was explicitly designed to host domain-specific languages [5]. It has been used to implement a variety of sublanguages, from a declarative syntax for probabilistic models to a BASIC interpreter [13, 23].

When using the AUTOMAN DSL, programmers first create an `AutoMan` instance, specifying a backend adapter, which indicates the crowdsourcing system that should be used (e.g., Mechanical Turk) and how it should be configured (e.g., user credentials, etc.). Next, the `Question` function is declared with the desired statistical `confidence` level and any other crowdsourcing backend-specific task parameters as required. When programmers call their `Question` function with some input data, the AUTOMAN scheduler launches the task asynchronously, allowing the main program to continue while the slower human computation runs in the background.

Since our aim was to make task specification simple, and to automate as many functions as possible, our Mechanical Turk compatibility layer provides sane defaults for many of the parameters. Additionally, we delegate control of task timeouts and rewards to AUTOMAN, which will automatically adjust them to incentivize workers. Maximizing automation allows for concise task specification for the common cases. When our defaults are not appropriate for a particular program, the programmer may override them.

6.2 Abstract Questions and Concrete Questions

The main purpose of the DSL is to help programmers construct `Question` objects, which represent the system's promise to return a scalar `Answer` to the end-user. In reality, many concrete instantiations of this question, which we call `ConcreteQuestions`, may be created in the process of computing a single `Question`. The details of interacting with third-party crowdsourcing backends is handled by the `AutomanAdapter` layer, which describes how `ConcreteQuestions` are marshalled.

AUTOMAN controls scheduling of all `ConcreteQuestions` in the target crowdsourcing system. After programmers have defined a `Question`, they can then call the resulting object as if it were a standard programming language function. In other words, they provide input as arguments to the function, and receive output as a return value from the function, which can be fed as input to other tasks as desired.

From this point on, AUTOMAN handles communication with the crowdsourcing backend, task scheduling, quality control, and returning a result back to the programmer under budget and in a timely manner. `Question` threads are implemented using Scala Futures. After calling a `Question`, program control returns to the calling function, and execution of the human function proceeds in the background, in parallel.

The outputted `Future[Answer]` object is available to use immediately, and may be passed as input to other `Question` function calls. If the `Future[Answer]`'s value is read, it will block until the runtime completes computation and propagates values into those objects. Any number of `Questions` may run concurrently, subject only to the limitations of the underlying backend.

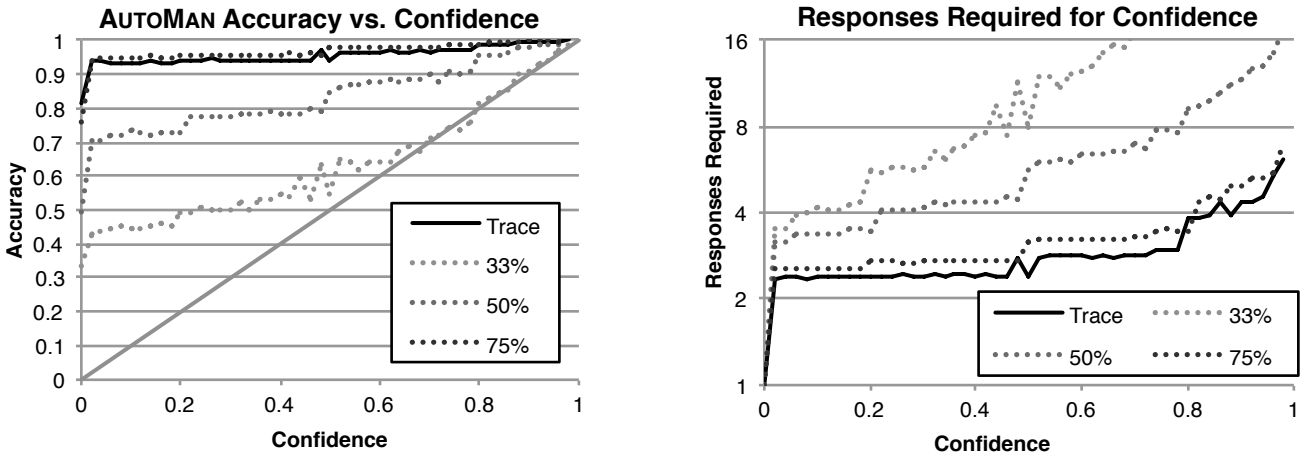


Figure 6. 10000 random sequences of worker responses to a five-option radio button question were used to simulate AUTOMAN's quality control algorithm at each confidence value. The trace lines use real worker responses to the "Which one does not belong?" application described in Section 7.1, while the 33%, 50%, and 75% lines were generated from synthetic traces where 33%, 50%, and 75% of workers chose the correct response, respectively. These graphs show that AUTOMAN is able to maintain the accuracy of final answers even when individual workers have low accuracy. Increasing confidence and decreasing worker accuracy both lead to exponential growth in the number of responses required to select a final result.

6.3 Memoization of Results

AUTOMAN's automatic memoization stores `Answer` data in a lightweight Apache Derby database. Implementors of third party `AutomanAdapters` must provide a mapping between their concrete `Answer` representation and AUTOMAN's canonical form for `Answer` data.

If a program halts abnormally, when that program is resumed, AUTOMAN first checks the memoization database for answers matching the program's `Question` signature before attempting to schedule more tasks. If a programmer changes the `Question` before restarting the program, this signature will no longer match, and AUTOMAN will treat the `Question` as if it had never been asked. Any future use of a memoized function amortizes the initial cost of the function by reusing the stored value, and as long as the programmer preserves the memo database, reuse of memoized functions works across program invocations, even for different programs.

It is incumbent on the user to ensure that they define side-effect-free AUTOMAN functions. Scala does not currently provide a keyword to enforce functional purity.

6.4 Validation strategies

The manner in which jobs are scheduled and errors handled depends on the chosen `ValidationStrategy`. If a strategy is not specified, AUTOMAN automatically uses the validation routines in the `DefaultStrategy`, which performs the form of statistical error handling we outlined in earlier sections. However, in the event that more sophisticated error handling is required, the programmer may either extend or completely replace our base error-handling strategy by implementing the `ValidationStrategy` interface.

6.5 Third-party implementors

Implementors who wish to adapt the AUTOMAN runtime for additional crowdsourcing backends need only implement the `AutomanAdapter` and `ConcreteQuestion` interfaces. Programs for one crowdsourcing backend thus can be ported to a new system by including the appropriate `AutomanAdapter` library and specifying the proprietary system's configuration details.

7. Evaluation

We implemented three sample applications using AUTOMAN: a semantic image-classification task using checkboxes, an image-counting task using radio buttons, and an optical character recognition (OCR) pipeline. These applications were chosen to be representative of the kinds of problems which remain difficult even for state-of-the-art algorithms.

7.1 Which one does not belong?

Our first sample application asks users to identify which object does not belong in a collection of items (Figure 1). This kind of task requires both image- and semantic-classification capability, and is a component in clustering and automated construction of ontologies. Because tuning of AUTOMAN's parameters is largely unnecessary, relatively little code is required to implement this functionality (about 20 lines).

We gathered 93 responses from workers during our sampling runs. Runtimes for this program were on the order of minutes, but there is substantial variation in runtime given

```
import com.amazonaws.services.s3.AmazonS3Client
import java.awt.image.BufferedImage
import java.io.File
import edu.umass.cs.automan.adapters.MTurk._

object HowManyThings {
  def main(args: Array[String]) {
    val a = MTurkAdapter { mt =>
      mt.access_key_id = "XXXX"
      mt.secret_access_key = "XXXX"
      mt.sandbox_mode = true
    }

    def how_many(url:String)=a.RadioButtonQuestion{q=>
      q.text="How many "+args(0)+" are in this image?"
      q.image_url = url
      q.options = List(
        a.Option('zero, "None"),
        a.Option('one, "One"),
        a.Option('more, "More than one")
      )
    }

    // Search for a bunch of images
    val urls = get_urls(args(0))

    // download each image
    val images = urls.map(download_image(_))

    // resize each image
    val scaled = images.map(resize(_))

    // store each image in S3
    val s3client = init_s3()
    val s3_urls = scaled.map{ i =>
      store_in_s3(i, s3client)
    }

    // ask humans for answers
    val answers_urls = s3_urls.map { url =>
      (how_many(getTinyURL(url.toString)) -> url)
    }

    // print answers
    answers_urls.foreach { case(a,url) =>
      println("url: " + url +
              ", answer: " + a().value)
    }
  }
}
```

Figure 7. An application that counts the number of searched-for objects in the image. Amazon S3, Google, TinyURL, and image-manipulation helper functions have been omitted.

the time of the day. Demographic studies of Mechanical Turk have shown that the majority of workers on Mechanical Turk are located in the United States and in India [16]. These findings largely agree with our experience, as we found that this program (and variants) took upward of several hours during the late evening hours in the United States.

Results from this application were used to test AUTOMAN's quality control algorithm at different confidence levels. AUTOMAN ensures that each worker's response to this application is independent of all other responses. Because responses are independent, we can shuffle the order of responses and re-run AUTOMAN to approximate many different runs of the same application with the same worker accuracy. Figure 6 shows the average accuracy of AUTOMAN's final answer at each confidence level, and the average number of responses required before AUTOMAN was confident in the final answer. Figure 6 also includes results for three sets of synthetic responses. Synthetic traces are generated by returning a correct answer with probability equal to the specified worker accuracy (33%, 50%, and 75%). Incorrect answers are uniformly distributed over the four remaining choices.

These results show that AUTOMAN's quality control is highly pessimistic. Even with extremely low worker accuracy, AUTOMAN is able to maintain high accuracy of final results. Real worker responses are typically quite accurate (over 80% in this case), and AUTOMAN rarely needs to exceed the first two rounds of three questions to reach a very high confidence.

7.2 How many items are in this picture?

Counting the number of items in an image also remains difficult for state-of-the-art machine learning algorithms. Machine-learning algorithms must integrate a variety of feature detection and contextual reasoning algorithms in order to achieve a fraction of the accuracy of human classifiers [26]. Moreover, vision algorithms that work well for all objects remain elusive.

This kind of task is trivial in AUTOMAN. We set up an image processing pipeline using the code in Figure 7. This application takes a search string as input, downloads images using Google Image Search, resizes the images, uploads the images to Amazon S3, ambiguates the URLs using TinyURL, and then posts the question "How many $items are in this image?"

We ran this task 8 times, spawning 71 question instances, and employing 861 workers, at the same time of the day (10 a.m. EST). AUTOMAN ensured that for each of the 71 questions asked, a worker was not able to participate more than once. We found that the mean runtime was 8 minutes, 20 seconds and that the median runtime was 2 minutes, 35 seconds. Overall, the typical task latency was surprisingly short.

The mean is skewed upward by the presence of one long-running task which asked "How many spoiled apples are in this image?". The difference of opinion caused by the ambiguity of the word "spoiled" caused worker answers to be nearly evenly distributed between two answers. This ambiguity forced AUTOMAN to collect a large number of responses to be able to meet the desired confidence level. AUTOMAN handled this unexpected behavior correctly, running until statistical confidence was reached.

7.3 Automatic number plate recognition (ANPR)

Our last application is a reimplementation of a common—and controversial—image recognition algorithm for automatic number plate recognition (ANPR). ANPR is widely deployed using distributed networks of traffic cameras. Academic literature on the subject suggests that state-of-the-art systems can achieve accuracy near 90% under ideal conditions [11]. False positives can have dramatic negative consequences in unsupervised ANPR systems as tickets are issued to motorists

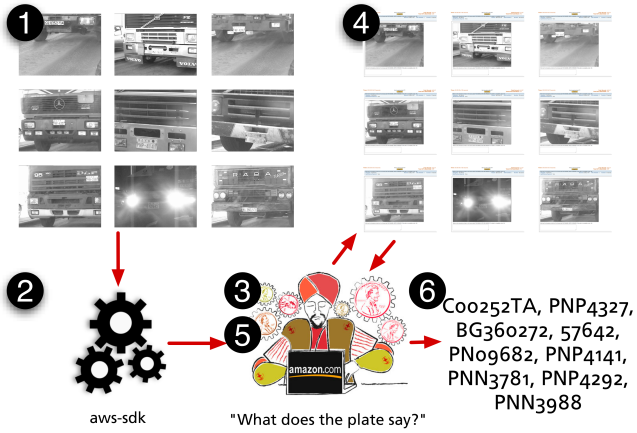
```
import collection.mutable
import com.amazonaws.services.s3.AmazonS3Client
import edu.umass.cs.automan.adapters.MTurk._

object Anpr extends App {
  val opts = optparse(args)

  val a = MTurkAdapter {
    mt =>
      mt.access_key_id = opts('key)
      mt.secret_access_key = opts('secret)
      mt.sandbox_mode = false
  }

  def plate_text(url:String)=a.FreeTextQuestion {q=>
    q.budget = 5.00
    q.text = "What are the characters printed on " +
             "this license plate? NO DASHES, DOTS" +
             " OR SPACES! If the plate is " +
             "unreadable, enter: NA"
    q.image_url = url
    q.allow_empty_pattern = true
    q.pattern = "XXXXXYYY"
  }

  // store each image in S3
  val filehandles = get_fh(opts('directory))
  val s3client: AmazonS3Client = init_s3()
  val s3_urls = filehandles.map { fh =>
    store_in_s3(fh, s3client)
  }

  // get plate texts for the good ones
  val plate_texts = s3_urls.par.map { url =>
    plate_text(url)().value
  }

  // print out results
  plate_texts.foreach { text => println(text) }
}
```

Figure 8. A number-plate recognition program. Amazon S3, command-line option parsing, and other helper functions have been omitted for clarity.

automatically. A natural consequence of this fact is the need for human supervision to audit results and limit false positives.

Figure 8 shows an ANPR application written in AUTOMAN. We evaluated this application using the MediaLab LPR database[3]. The benchmark was run twice on 72 of the "extremely difficult" images, for a total of 144 license plate identifications. Overall accuracy was 91.6% for this subset. Each task cost an average of 12.08 cents, with a median latency of just under 2 minutes per image. AUTOMAN runs all identification tasks in parallel: one complete run took just under three hours, while the other took less than one hour. These translate to throughputs of 24 and 69 plates/hr. While the AUTOMAN application is slower than computer vision approaches, it could be used for only the most difficult images to increase accuracy at low cost.

Automatic number plate recognition is a long-studied problem: a recent survey has 49 citations spanning 21 years [11]. Using AUTOMAN to engage humans to perform this task required only a few hours of programming time and AutoMan's



(a) Visual representation of the license-plate-recognition workflow. 1) Images are read from disk. 2) Images are uploaded to Amazon S3. 3) HITs are posted to MTurk by AUTOMAN. 4) Workers complete the posted HITs. 5) Responses are gathered by AUTOMAN. 6) AUTOMAN chooses the correct answers, repeating steps 3-5 as necessary, and prints them to the screen.

(b) A sample HIT on Mechanical Turk for OCR. In all of our trial runs, AUTOMAN correctly identified this hard-to-read plate.

Figure 9. An AUTOMAN program for automatic number plate recognition.

quality control ensures that it delivers results that match or exceed the state-of-the-art on even the most difficult cases.

8. Related Work

Example Uses of Crowdsourcing. The computational power of many software applications is significantly enhanced with human supervision at critical steps. The innate ability of humans to quickly and correctly provide answers to otherwise intractable problems has resulted in a great deal of interest in hybrid human-computer applications. We describe a representative sample of such applications below. Note

Feature	AUTOMAN	TurkIt [21]	CrowdFlower [25]	Jabberwocky [1]	Turkomatic [19]	CrowdDB [14]
Quality Control Guarantee	✓					
High Performance	✓					
Automatic Budgeting	✓					
Automatic Time Optimization	✓					
Automatic Task Accept/Reject	✓					
General Purpose	✓	✓	✓	✓	✓	
Memoizes Results	✓	✓				
Interfaces with Existing Code	✓Java	✓Javascript		✓Ruby		✓SQL
Type-Safe	✓					
Platform Agnostic	✓		*n/a*	✓		

Table 1. A comparison between AUTOMAN and other crowdsourcing-based systems.

that these do not rely on general-purpose crowdsourcing platforms built on open labor markets, but simply involve humans in computations.

Galaxy Zoo uses humans as image classifiers to determine the direction of the spin of distant galaxies [20]. Participants, who have little training, provide image classifications (called "votes") which are weighted by their accuracy relative to other participants.

reCAPTCHA repurposes the popular CAPTCHA web-based "human Turing test" to classify images of text from scanned books where traditional optical-character recognition algorithms have failed [29]. Since deploying the software, reCAPTCHA has recognized millions of words with an estimated average accuracy exceeding 99%.

CrowdSearch relies on human judgment to improve the accuracy of web-based mobile phone image search, in near real-time [30]. Human workers are recruited iteratively and in batches, according to arrival-rate estimates and a majority-voting scheme.

FoldIt! presents difficult protein-folding problems as an online video game. Participants collaborate with state-of-the-art protein-folding models to search the state space of protein configurations faster than unsupervised algorithms [6].

Programming the Crowd. While there has been substantial *ad hoc* use of crowdsourcing platforms, especially Amazon's Mechanical Turk, there has been little effort to manage workers programatically. Amazon's Mechanical Turk exposes a low-level API limited to allowing jobs to be submitted, tracked, and checked.

TurKit is a scripting system designed to make it easier to manage Mechanical Turk tasks [21]. TurKit Script extends JavaScript with a templating feature for common Mechanical Turk tasks, and adds checkpointing to avoid re-submitting Mechanical Turk tasks if a script fails. CrowdForge is a web tool that wraps a MapReduce-like abstraction on Mechanical Turk tasks [8, 18]. Programmers decompose tasks into *partition* tasks, *map* tasks, and *reduce* tasks. CrowdForge automatically handles distributing tasks to multiple users and collecting the results. Unlike AUTOMAN, neither TurKit nor CrowdForge automatically manage scheduling, pricing, or quality control; in addition, TurkIt's embedding in JavaScript also limits its usefulness for compute-intensive tasks.

CrowdDB models crowdsourcing as an extension to relational databases, providing annotations to traditional SQL queries that trigger the SQL runtime to crowdsource database cleansing tasks [14]. The SQL runtime is crowdsourcing-aware, so that the SQL's query planner can minimize operations that would otherwise be very expensive. Unlike AUTOMAN, CrowdDB is not a general platform for computing, and relies on majority voting as its sole quality control mechanism.

Turkomatic aims to crowdsource an entire computation, including the "programming" of the task [19]. Tasks are provided to the system in plain English, and the Turkomatic runtime proceeds in two steps: a map step and a reduce step. In the map step, workers provide an execution plan, which is then carried out in the reduce step. Like AUTOMAN, Turkomatic can be used to construct arbitrarily complex computations. However, Turkomatic does not handle budgeting or quality control, and also cannot be integrated with a conventional programming language.

Jabberwocky provides a human-computation stack on top of a MapReduce-like parallel programming framework called ManReduce [1]. A Ruby DSL called Dog eases programmer interaction with a resource-management layer called Dormouse, which does the actual scheduling of tasks on ManReduce. Jabberwocky uses a fixed, optional quality control scheme based on majority voting and a fixed pricing scheme.

Quality Control. CrowdFlower is a closed-source web service that targets commercial crowdsourcing platforms [25]. To enhance quality, CrowdFlower uses a "gold-seeding" approach to identify likely erroneous workers, sprinkling questions with known answers into the question pipeline. CrowdFlower incorporates methods to programmatically generate this data via "fuzzing" as the system processes real work, in an effort to ease the gold-generation burden on the requester. Recognizing new types of errors remains a manual process. Like other work in this area, this approach focuses on establishing trust in the quality of a particular worker [17]. Rather

than trust that one can extrapolate quality of work on a new task from a worker's past performance, AUTOMAN addresses work quality directly.

Shepherd provides interactive feedback between task requesters and task workers in an effort to increase quality; the idea is to *train* workers to do a particular job well [10]. This approach requires ongoing interaction between requesters and workers, while AUTOMAN requires none.

Soylent introduces the *find-fix-verify* pattern of quality control for written documents. The idea is to crowdsource three distinct phases: finding errors, fixing errors, and verifying the fixes [4]. Soylent can handle open-ended questions, which AUTOMAN currently does not support. However, unlike AUTOMAN, Soylent's approach does not provide any quantitative guarantees about the quality of the output.

9. Future Work

We plan to build on the existing AUTOMAN prototype in the following directions:

Broader question classes. The current AUTOMAN system supports radio-button, checkbox, and restricted free-text questions. We plan to extend AUTOMAN to support questions with unrestricted free-text answers, i.e., open-ended questions. The validation strategy will add an intermediate step that depends on workers to rank answers, and then perform quality control on the rankings.

Additional automatic tuning. Currently, AUTOMAN cannot distinguish between the scenario where there is a low arrival rate of workers because we do not offer a compelling incentive and the scenario where workers are simply not available due to the time of day. In the latter case, we do not want to increment incentives, because doing so will have no effect. We plan to investigate methods to adjust this behavior in a future version.

Visualization tools. While AUTOMAN hides the management details of human-based computation beneath an abstraction layer, it can be useful for debugging to peel back this layer to see how tasks are progressing. AUTOMAN currently provides a simple logging mechanism, but as the number of jobs becomes large, navigating logs quickly becomes onerous. Building on the fact that the AUTOMAN runtime system already acts as a server, we plan to extend it with a web service that will allow AUTOMAN programmers to view the jobs in the system. We are initially planning to include visualizations of the execution graph (including summaries) and allow searching for jobs matching certain criteria.

Event simulator. Debugging AUTOMAN applications currently requires running one's program with the `sandbox_mode` flag set, which will post jobs to Mechanical Turk's sandbox website for developers. While this is handy for previewing a task, it is less than ideal to drive the behavior of the entire program, since the number of independent workers required to participate in a computation can be substantial. Other crowdsourcing backends may also lack this sandbox feature. We plan to build both an event simulator that can generate worker responses drawn from arbitrary distributions, and an application trace replayer, which can simulate the backend using real data.

10. Conclusion

Humans can perform many tasks with ease that remain difficult or impossible for computers. This paper presents AUTOMAN, the first *crowdprogramming* system. Crowdprogramming integrates human-based and digital computation. By automatically managing quality control, scheduling, and budgeting, AUTOMAN allows programmers to easily harness human-based computation for their applications.

AUTOMAN is available for download at `http://www.automan-lang.org`.

Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. CCF-1144520 and DARPA Award N10AP2026. The fourth author is supported by the National Science Foundation under Grant No. CCF-0953754. The authors gratefully acknowledge Mark Corner for his support for this work and initially prompting us to explore the area of crowdsourcing. We also thank the anonymous reviewers for their helpful comments.

References

[1] S. Ahmad, A. Battle, Z. Malkani, and S. Kamvar. The Jabberwocky Programming Environment for Structured Social Computing. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology*, UIST '11, pages 53–64, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0716-1. URL `http://doi.acm.org/10.1145/2047196.2047203`.

[2] Amazon. Mechanical Turk. `http://www.mturk.com`, June 2011.

[3] C. N. Anagnostopoulos. MediaLab LPR Database.

[4] M. S. Bernstein, G. Little, R. C. Miller, B. Hartmann, M. S. Ackerman, D. R. Karger, D. Crowell, and K. Panovich. Soylent: A Word Processor with a Crowd Inside. In K. Perlin, M. Czerwinski, and R. Miller, editors, *UIST*, pages 313–322. ACM, 2010. ISBN 978-1-4503-0271-5. URL `http://dblp.uni-trier.de/db/conf/uist/uist2010.html#BernsteinLMHAKCP10`.

[5] H. Chafi, Z. DeVito, A. Moors, T. Rompf, A. K. Sujeeth, P. Hanrahan, M. Odersky, and K. Olukotun. Language Virtualization for Heterogeneous Parallel Computing. In *Onward!*, 2010.

[6] S. Cooper, F. Khatib, A. Treuille, J. Barbero, J. Lee, M. Beenen, A. Leaver-Fay, D. Baker, Z. Popovi, and F. Players. Predicting protein structures with a multiplayer online game. *Nature*, 466(7307):756–760, 2010. URL `http://www.nature.com/doifinder/10.1038/nature09304`.

[7] A. DasGupta. *Probability for Statistics and Machine Learning: Fundamentals and Advanced Topics*. Springer, 1st edition, 2011.

[8] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*, pages 137–150, 2004.

[9] J. R. Douceur. The Sybil Attack. In *Revised Papers from the First International Workshop on Peer-to-Peer Systems*, IPTPS '01, pages 251–260, London, UK, 2002. Springer-Verlag. ISBN 3-540-44179-4. URL http://dl.acm.org/citation.cfm?id=646334.687813.

[10] S. Dow, A. Kulkarni, B. Bunge, T. Nguyen, S. Klemmer, and B. Hartmann. Shepherding the Crowd: Managing and Providing Feedback to Crowd Workers. In *Proceedings of the 2011 Annual Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 1669–1674, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0268-5. URL http://doi.acm.org/10.1145/1979742.1979826.

[11] S. Due, M. Ibrahim, M. Shehata, and W. Badawy. Automatic License Plate Recognition (ALPR): A State of the Art Review. *IEEE Transactions on Circuits and Systems for Video Technology*, PP, 2012.

[12] W. Feller. *An Introduction to Probability Theory and Applications*, volume 1. John Wiley & Sons Publishers, 3rd edition, 1968.

[13] M. Fogus. BAYSICK—a DSL for Scala implementing a subset of BASIC. https://github.com/fogus/baysick, March 2009.

[14] M. J. Franklin, D. Kossmann, T. Kraska, S. Ramesh, and R. Xin. CrowdDB: Answering Queries with Crowdsourcing. In T. K. Sellis, R. J. Miller, A. Kementsietsidis, and Y. Velegrakis, editors, *SIGMOD Conference*, pages 61–72. ACM, 2011. ISBN 978-1-4503-0661-4.

[15] J. Howe. The Rise of Crowdsourcing. *Wired Magazine*, 14(6): 176–178, 2006. ISSN 1059-1028.

[16] P. G. Ipeirotis. Demographics of Mechanical Turk. Technical Report Working Paper CeDER-10-01, NYU Center for Digital Economy Research, 2010.

[17] P. G. Ipeirotis, F. Provost, and J. Wang. Quality management on Amazon Mechanical Turk. In *Proceedings of the ACM SIGKDD Workshop on Human Computation*, HCOMP '10, pages 64–67, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0222-7. doi: http://doi.acm.org/10.1145/1837885.1837906. URL http://doi.acm.org/10.1145/1837885.1837906.

[18] A. Kittur, B. Smus, and R. E. Kraut. CrowdForge: Crowdsourcing Complex Work. Technical Report CMU-HCII-11-100, Human-Computer Interaction Institute, School of Computer Science, Carnegie Mellon University, February 2011.

[19] A. P. Kulkarni, M. Can, and B. Hartmann. Turkomatic: Automatic Recursive Task and Workflow Design for. Mechanical Turk. In *Proceedings of the 2011 Annual Conference Extended Abstracts on Human Factors in Computing Systems*, CHI EA '11, pages 2053–2058, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0268-5. URL http://doi.acm.org/10.1145/1979742.1979865.

[20] K. Land, A. Slosar, C. Lintott, D. Andreescu, S. Bamford, P. Murray, R. Nichol, M. J. Raddick, K. Schawinski, A. Szalay, D. Thomas, and J. Vandenberg. Galaxy Zoo: the large-scale spin statistics of spiral galaxies in the Sloan Digital Sky Survey. *Monthly Notices of the Royal Astronomical Society*, 388:1686–1692, Aug. 2008. doi: 10.1111/j.1365-2966.2008.13490.x.

[21] G. Little, L. B. Chilton, M. Goldman, and R. C. Miller. TurKit: Human Computation Algorithms on Mechanical Turk. In *UIST*, pages 57–66, 2010.

[22] M. Marge, S. Banerjee, and A. Rudnicky. Using the Amazon Mechanical Turk for Transcription of Spoken Language. In *The 2010 IEEE International Conference on Acoustics Speech and Signal Processing (ICASSP)*, pages 5270–5273, Mar. 2010. doi: 10.1109/ICASSP.2010.5494979.

[23] A. McCallum, K. Schultz, and S. Singh. FACTORIE: Probabilistic Programming via Imperatively Defined Factor Graphs. In *Neural Information Processing Systems (NIPS)*, 2009.

[24] M. Odersky and M. Zenger. Scalable Component Abstractions. In *Proceedings of the 20th annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 41–57, New York, NY, USA, 2005. ACM. ISBN 1-59593-031-0. URL http://doi.acm.org/10.1145/1094811.1094815.

[25] D. Oleson, V. Hester, A. Sorokin, G. Laughlin, J. Le, and L. Biewald. Programmatic Gold: Targeted and Scalable Quality Assurance in Crowdsourcing. In *HCOMP '11: Proceedings of the Third AAAI Human Computation Workshop*. Association for the Advancement of Artificial Intelligence, 2011.

[26] D. Parikh and L. Zitnick. Human-Debugging of Machines. In *Second Workshop on Computational Social Science and the Wisdom of Crowds*, NIPS '11, 2011. URL http://www.cs.umass.edu/%7Ewallach/workshops/nips2011css/.

[27] D. Shahaf and E. Amir. Towards a theory of AI completeness. In *Commonsense 2007: 8th International Symposium on Logical Formalizations of Commonsense Reasoning*. Association for the Advancement of Artificial Intelligence, 2007.

[28] D. Tamir, P. Kanth, and P. Ipeirotis. Mechanical Turk: Now with 40.92% spam. http://www.behind-the-enemy-lines.com/2010/12/mechanical-turk-now-with-4092-spam.html, December 2010.

[29] L. von Ahn, B. Maurer, C. Mcmillen, D. Abraham, and M. Blum. reCAPTCHA: Human-Based Character Recognition via Web Security Measures. *Science*, 321(5895):1465–1468, August 2008. URL http://dx.doi.org/10.1126/science.1160379.

[30] T. Yan, V. Kumar, and D. Ganesan. CrowdSearch: Exploiting Crowds for Accurate Real-Time Image Search on Mobile Phones. In *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, pages 77–90, New York, NY, USA, 2010. ACM. ISBN 978-1-60558-985-5. URL http://doi.acm.org/10.1145/1814433.1814443.