DOI:10.1145/2927928

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

By Daniel W. Barowy, Charlie Curtsinger, Emery D. Berger, and Andrew McGregor

Abstract

Humans can perform many tasks with ease that remain difficult or impossible for computers. Crowdsourcing platforms like Amazon Mechanical Turk make it possible to harness human-based computational power at an unprecedented scale, but their utility as a general-purpose computational platform remains limited. The lack of complete automation makes it difficult to orchestrate complex or interrelated tasks. Recruiting 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.

We introduce AutoMan, the first fully automatic *crowd-programming* system. AutoMan integrates human-based computations into a standard programming language as ordinary function calls that 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.

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.[16, 18] Many researchers expect these "AI-complete" tasks to remain beyond the reach of computers for the foreseeable future.[19] Harnessing human-based computation in general and at scale faces the following challenges:

Determination of pay and time for tasks. Employers must decide the payment and time allotted before posting tasks. It is both difficult and important to choose these correctly since workers will not accept tasks with too-short deadlines or too little pay.

Scheduling complexities. Employers must manage the tradeoff 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 majority voting is neither necessary nor sufficient. In some cases, majority vote is too conservative, and in other cases, it is likely that workers will agree by chance.

Contributions

We introduce AutoMan, a programming system that integrates human-based and digital computation. AutoMan addresses the challenges of harnessing human-based computation at scale:

Transparent integration. AutoMan abstracts human-based computation as ordinary function calls, freeing the programmer from scheduling, budgeting, and quality control concerns (Section 3).

Automatic scheduling and budgeting. The AutoMan runtime system schedules tasks to maximize parallelism across human workers while staying under budget. AutoMan tracks job progress, reschedules, and reprices failed tasks as necessary (Section 4).

Automatic quality control. The AutoMan runtime system manages quality control automatically. AutoMan creates enough human tasks for each computation to achieve the confidence level specified by the programmer (Section 5).

2. BACKGROUND

Since crowdsourcing is a novel application domain for programming language research, we summarize the necessary background on crowdsourcing platforms. We focus on Amazon Mechanical Turk[a] (MTurk), but other crowdsourcing platforms are similar. MTurk acts as an intermediary between *requesters* and *workers* for short-term tasks.

Human intelligence task. In MTurk parlance, tasks are known as *human intelligence tasks* (HITs). Each HIT is

[a] Amazon Mechanical Turk is hosted at http://mturk.com.

The original version of this paper was published in the *Proceedings of OOPSLA 2012*.

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 either free-text questions, where workers provide a free-form textual response, or multiple-choice questions, where workers make one or more selections from a set of options. Most HITs on MTurk are for relatively simple tasks, such as "does this image match this product?" Compensation is generally low (usually a few cents) since employers expect that work to be completed quickly (on the order of seconds).

Requesting work. Requesters can create HITs using either MTurk's website or programmatically, using an API. Specifying a number of *assignments* greater than one allows multiple unique workers to complete the same task, parallelizing HITs. Distinct HITs with similar qualities can also be grouped to make it easy for workers to find similar work.

Performing work. Workers may choose any available task, subject to qualification requirements (see below). When a worker selects a HIT, she is granted a time-limited *reservation* for that particular piece of work such that no other worker can accept it.

HIT expiration. HITs have two timeout parameters: the amount of time that a HIT remains visible on MTurk, known as the *lifetime* of a HIT, and the amount of time that a worker has to complete an assignment once it is granted, known as the *duration* of an assignment. If a worker exceeds the assignment's duration without submitting completed work, the reservation is cancelled, and the HIT becomes available to other workers. If a HIT reaches the end of its lifetime without its assignments having been completed, the HIT expires and is made unavailable.

Requesters: Accepting or rejecting work. Once a worker submits a completed assignment, the requester may then accept or reject the completed work. Acceptance indicates that the completed work is satisfactory, at which point the worker is paid. Rejection withholds payment. The requester may provide a textual justification for the rejection.

Worker quality. The key challenge in automating work in MTurk is attracting good workers and discouraging bad workers from participating. MTurk provides no mechanism for requesters to seek out specific workers (aside from emails). Instead, MTurk provides a *qualification* mechanism that limits which workers may participate. A common qualification is that workers must have an overall assignment-acceptance rate of 90%.

Given the wide variation in tasks on MTurk, overall worker accuracy is of limited utility. For example, a worker may be skilled at audio transcription tasks and thus have a high accuracy rating, but it would be a mistake to assume on the basis of their rating that the same worker could also perform Chinese-to-English translation tasks. Worse, workers who cherry-pick easy tasks and thus have high accuracy ratings may be less qualified than workers who routinely perform difficult tasks that are occasionally rejected.

3. OVERVIEW

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

3.1. Using AutoMan

Figure 1 presents a real AUTOMAN program that recognizes automobile license plate texts from images. Note that the programmer need not specify details about the chosen crowdsourcing backend (Mechanical Turk) other than the appropriate backend adapter and account credentials. Crucially, all details of crowdsourcing are hidden from the AUTOMAN programmer. The AUTOMAN runtime abstracts away platform-specific interoperability code, schedules and determines budgets (both cost and time), and automatically ensures that outcomes meet a minimum confidence level.

Initializing AutoMan. After importing the AUTOMAN and MTurk adapter libraries, the first thing an AUTOMAN programmer does is to declare a configuration for the desired crowdsourcing platform. The configuration is then bound to an AUTOMAN runtime object that instantiates any platform-specific objects.

Specifying AutoMan functions. AUTOMAN functions declaratively describe questions that workers must answer. They must include the question type and may also include text or images.

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%, but this can be overridden as needed. The meaning and derivation of confidence is discussed in Section 5.

Metadata and question text. Each question declaration requires a title and description, used by the crowdsourcing

Figure 1. A license plate recognition program written using AUTOMAN. `getURLsFromDisk()` is omitted for clarity. 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).

```
import edu.umass.cs.automan.adapters.MTurk._

object ALPR extends App {
  val a = MTurkAdapter { mt =>
    mt.access_key_id = "XXXX"
    mt.secret_access_key = "XXXX"
  }

  def plateTxt(url:String) = a.FreeTextQuestion { q =>
    q.budget = 5.00
    q.text = "What does this license plate say?"
    q.image_url = url
    q.allow_empty_pattern = true
    q.pattern = "XXXXXYYY"
  }

  automan(a) {
    // get plate texts from image URLs
    val urls = getURLsFromDisk()
    val plate_texts = urls.map { url =>
      (url, plateTxt(url))
    }

    // print out results
    plate_texts.foreach { (url,outcome) =>
      outcome.answer match {
        case Answer(ans,_,_) =>
          println(url + ": "+ ans)
        case _ => ()
      }
    }
  }
}
```

platform's user interface. These fields map to MTurk's fields of the same name. A declaration also includes the question text itself, 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), where any number of answers may be correct ("checkbox" questions), and a restricted form of free-text entry. Section 5 describes how AUTOMAN's quality control algorithm handles each question type.

Invoking a function. A programmer can invoke an AUTOMAN function as if it were any ordinary (digital) function. In Figure 1, the programmer calls the `plateTxt` function with a URL pointing to an image as a parameter. The function returns an `Outcome` object representing a `Future[Answer]` that can then be passed as data to other functions. AUTOMAN functions execute eagerly, in a background thread, as soon as they are invoked. The program does not block until it needs to read an `Outcome.answer` field, and only then if the human computation is not yet finished.

4. SCHEDULING ALGORITHM

AUTOMAN's scheduler controls task marshaling, budgeting of time and cost, and quality. This section describes how AUTOMAN automatically determines these parameters.

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* that 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 g every time a task goes unanswered. g 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.
2. The reward should not grow so quickly that it incentivizes workers to wait for a larger reward.

Section 4.4 presents an analysis of reward growth rates. We also discuss the feasibility of our assumptions and possible attack scenarios in Section 5.4.

4.2. Scheduling the right number of tasks

AUTOMAN's default policy for spawning tasks is optimistic: it creates the smallest number of tasks required to reach the desired confidence level when workers agree unanimously. If workers do agree unanimously, AUTOMAN returns their answer. Otherwise, AUTOMAN computes and then schedules the minimum number of additional votes required to reach confidence.

When the user-specified budget is insufficient, AUTOMAN suspends the computation before posting additional tasks. The computation can either be resumed with an increased budget or accepted as-is, with a confidence value lower than the one requested. The latter case is considered exceptional, and must be explicitly handled by the programmer.

4.3. Trading off latency and money

AUTOMAN allows programmers to provide a *time-value* parameter that counterbalances the default optimistic assumption that all workers will agree. The parameter instructs the system to post more than the minimum number of tasks in order to minimize the latency incurred when jobs are serialized across multiple rounds. The number of tasks posted is a function of the *value of the programmer's time*:

$$\max\left(\text{min\_required}, \left\lfloor \frac{\text{time\_value}}{\text{min\_wage}} \right\rfloor\right).$$

As a cost savings, when AUTOMAN receives enough answers to reach the specified confidence, it cancels all unaccepted tasks. In the worst case, all posted tasks will be answered before AUTOMAN can cancel them, which will cost no more than time_value · task_timeout. While this strategy runs the risk of paying substantially more for a computation, it can yield dramatic reductions in latency. We re-ran the example program described in Section 7.1 with a time-value set to $50. In two separate runs, the computation completed in 68 and 168 *seconds*; by contrast, the default time-value (minimum wage) took between 1 and 3 *hours* to complete.

4.4. Maximum reward growth rate

When workers encounter a task with an initial reward of R they may choose to accept the task or wait for the reward to grow. If R is below R_{min}, the smallest reward acceptable to workers, then tasks will not be completed. Let g be the reward growth rate and let i be the number of discrete time steps, or *rounds*, that elapse from an initial time $i = 0$, such that a task's reward after i rounds is $g^i R$. We want a g large enough to reach R_{min} quickly, but not so large that workers have an incentive to wait. We balance the probability that a task remains available against the reward's growth rate so workers should not expect to profit by waiting.

Let p_a be the probability that a task remains available from one round to the next, assuming this probability is constant across rounds. Suppose a worker's strategy is to wait i rounds and then complete the task for a larger reward. The expected reward for this worker's strategy is

$$\mathbb{E}\left[\text{reward}_i\right] = (p_a g)^i R.$$

when $g \leq 1/p_a$, the expected reward is maximized at $i = 0$; workers have no incentive to wait, even if they are aware of AUTOMAN's pricing strategy. A growth rate of exactly $1/p_a$ will reach R_{min} as fast as possible without incentivizing waiting. This pricing strategy remains sound even when p_a is not constant, provided the desirability of a task does not decrease with a larger reward.

The true value of p_a is unknown, but it can be estimated by modeling the acceptance or rejection of each task as an independent Bernoulli trial. The maximum likelihood estimator

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

is a reasonable estimate for p_a, where n is the number of times a task has been offered and t is the number of times the task was not accepted before timing out. To be conservative, $\tilde{p}_a$ can be over-approximated, driving g downward. The difficulty of choosing a reward a priori is a strong case for automatic budgeting.

5. QUALITY CONTROL

AUTOMAN's quality control algorithm is based on collecting enough consensus for a given question to rule out the possibility, for a specified level of confidence, that the results are due to random chance. AUTOMAN's algorithm is adaptive, taking both the programmer's confidence threshold and the likelihood of random agreement into account. By contrast, majority rule, a commonly used technique for achieving higher-quality results, is neither necessary nor sufficient to rule out outcomes due to random chance (see Figure 2). A simple two-option question (e.g., "Heads or tails?") with three *random* respondents demonstrates the problem: a majority is not just likely, it is guaranteed. Section 5.4 justifies this approach.

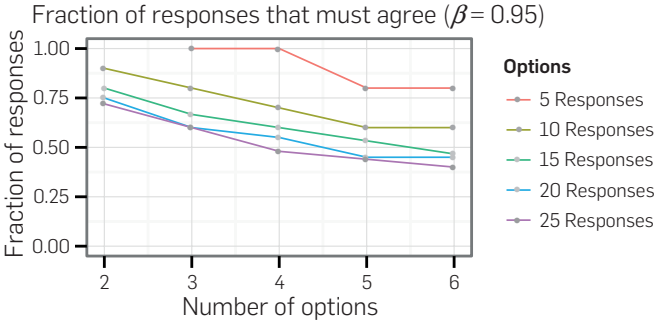
Initially, AUTOMAN spawns enough tasks to meet the desired confidence level if all workers who complete the tasks agree unanimously. Computing the confidence of an outcome in this scenario is straightforward. Let k be the number of options, and n be the number of tasks. The confidence is then $1 - k(1/k)^n$. AUTOMAN computes the smallest n such that the probability of random agreement is less than or equal to one minus the specified confidence threshold.

Humans are capable of answering a rich variety of question types. Each of these question types requires its own probability analysis.

○ **Radio** ● **Buttons** For multiple-choice "radio-button" questions where only one choice is possible, k is exactly the number of possible options.

☑ **Check** □ **Boxes** For "checkbox" questions with c boxes, k is much larger: $k = 2^c$. In practice, k is often large enough that as few as two workers are required to rule out random behavior. To avoid accidental agreement caused when low-effort workers simply submit a form without changing any of the checkboxes, AUTOMAN randomly pre-selects checkboxes.

Figure 2. The fraction of workers that must agree to reach 0.95 confidence for a given number of tasks. For a three-option question and 5 workers, 100% of the workers must agree. For a six-option question and 15 or more workers, only a plurality is required to reach confidence. Notice that majority vote is neither necessary nor sufficient to rule out random respondents.



Free-text Input Restricted "Free-text" input is mathematically equivalent to a set of radio-buttons where each option corresponds to a valid input string. Nonetheless, even a small set of valid strings represented as radio buttons would be burdensome for workers. Instead, workers are provided with a text entry field and the programmer supplies a pattern representing valid inputs so that AUTOMAN can perform its probability analysis.

AUTOMAN's pattern specification syntax resembles COBOL's picture clauses. `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. For example, a telephone number recognition application might use the pattern `09999999999`.

For example, given a 7-character numeric pattern with no optional characters, $k = 10^7$. Again, k is often large, so a small number of HITs suffice to achieve high confidence in the result. As with checkbox questions, AUTOMAN treats free-text questions specially to cope with low-effort workers who might simply submit an empty string. To avoid this problem, AUTOMAN only accepts the empty string if it is explicitly entered with the special string `NA`.

5.1. Definitions

Formally, AUTOMAN's quality control algorithm depends on two functions, t and v, and associated parameters β and p^*. t computes the minimum threshold (the number of votes) needed to establish that an option is unlikely to be due to random chance with probability β (the programmer's confidence threshold). t depends on the random variable X, which models when n respondents choose one of k options uniformly at random. If no option crosses the threshold, v computes the additional number of votes needed. v depends on the random variable Y, which models a worker choosing the correct option with the observed probability p^* and all other options uniformly at random.

Let X and Y be multinomial distributions with parameters $(n, 1/k, \ldots, 1/k)$ and $(n, p, q, \ldots, q)$, respectively, where $q = (1-p)/(k-1)$. We define two functions $\mathcal{E}_1$ and $\mathcal{E}_2$ that have the following properties[2]:

LEMMA 5.1.

$$\Pr\left[\max_i X_i < m\right] = \mathcal{E}_1(n, m),$$

$$\Pr\left[\max_{i \geq 2} Y_i < m \leq Y_1\right] = \mathcal{E}_2(p, n, m),$$

where

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

and

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

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

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

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

Thus, when when n voters each choose randomly, the probability that any option meets or exceeds the threshold $t(n, \beta)$ is at most $\alpha = 1 - \beta$.

Finally, we define v, the number of extra votes needed, as

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

If workers have a bias of at least p^* toward a "popular" option (the remaining options being equiprobable), then when we ask $v(p^*, \beta)$ voters, the number of votes cast for the popular option passes the threshold (and all other options are below threshold) with probability at least β.

5.2. Quality control algorithm

AUTOMAN's quality control algorithm, which gathers responses until it can choose the most popular answer not likely to be the product of random chance, proceeds as follows:

1. Set $b = \min\{m \mid t(m, \beta) \neq \infty\}$. Set $n = 0$.
2. Ask b workers to vote on a question with k options. Set $n = b + n$.
3. If any option has more than $t(n, \beta)$ votes, return the most frequent option as the answer.
4. Let $b = v(p^*, \beta)$ and repeat from step 2.

Figure 2 uses t to compute the smallest *fraction* of workers that need to agree for $\beta = 0.95$. As the number of tasks and the number of options increase, the proportion of workers needed to agree decreases. For example, for a 4-option question with 25 worker responses, only 48% (12 of 25) of workers must agree to meet the confidence threshold. This figure clearly demonstrates that quality control based on majority vote is neither necessary nor sufficient to limit outcomes based on random chance.

5.3. Multiple comparisons problem

Note that AUTOMAN must correct for a subtle bias that is introduced as the number of rounds—and correspondingly, the number of statistical tests—increases. This bias is called the *multiple comparisons problem*. As the number of hypotheses grows with respect to a fixed sample size, the probability that at least one true hypothesis will be incorrectly falsified by chance increases. Without the correction, AUTOMAN is susceptible to accepting low-confidence answers when the proportion of good workers is low. AUTOMAN applies a Bonferroni correction to its statistical threshold, which ensures that the *familywise error rate* remains at or below the $1 - \beta$ threshold set by the programmer.[10] We empirically evaluate the cost and time overhead for this correction in Section 7.4.

5.4. Quality control discussion

For AUTOMAN's quality control algorithm to work, two assumptions must hold: (1) workers must be independent, and (2) 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 by masquerading as multiple workers, performing multiple tasks, or by colluding on tasks. We address each scenario below.

Scenario 1: Sybil Attack. A single user who creates multiple electronic identities for the purpose of thwarting identity-based security policy is known in the literature as a "Sybil attack."[6] The practicality of a Sybil attack depends directly on the feasibility of generating multiple identities.

Carrying out a Sybil attack on MTurk would be burdensome. Since MTurk 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 to forge.

Scenario 2: One Worker, Multiple Responses. In order increase the pay or allotted time for a task, MTurk requires requesters to post a new HIT. This means that a single AUTOMAN task can span multiple MTurk HITs. MTurk provides a mechanism to ensure worker uniqueness for a single HIT that has multiple assignments, but it lacks the functionality to ensure that worker uniqueness is maintained across multiple HITs. For AUTOMAN's quality control algorithm to be effective, AUTOMAN must be certain that workers who previously supplied responses cannot supply new responses for the same task.

Our workaround for this shortcoming is to use MTurk's "qualification" feature inversely: once a worker completes a HIT, AUTOMAN grants the worker a special "disqualification" that precludes them from supplying future responses.

Scenario 3: Worker Collusion. While it is appealing to lower the risk of worker collusion by ensuring that workers are geographically separate (e.g., by using IP geolocation), eliminating this scenario entirely is not practical. Workers can collude via external channels (e-mail, phone, word-of-mouth) to thwart our assumption of independence. Instead, we opt to make the effort of thwarting defenses undesirable given the payout.

By spawning large numbers of tasks, AUTOMAN makes it difficult for any group of workers to monopolize them. Since MTurk hides the true number of assignments for a HIT, workers cannot know how many wrong answers are needed to defeat AUTOMAN's quality control algorithm. This makes collusion infeasible. The bigger threat comes from workers who do as little work as possible to get compensated: previous research on MTurk suggests that random-answer spammers are the primary threat.[20]

Random as worst case AUTOMAN's quality control algorithm is based on excluding random responses. AUTOMAN gathers consensus not just until a popular answer is revealed, but also until its popularity is unlikely to be the product of random chance. As long as there is a crowd bias toward

the correct answer, AUTOMAN's algorithm will eventually choose it. Nevertheless, it is possible that workers could act maliciously and deliberately choose incorrect answers.

Random choice is a more realistic worst-case scenario: participants have an economic incentive not to deliberately choose incorrect answers. First, a correct response to a given task yields an immediate monetary reward. If workers know the correct answer, it is against their own economic self-interest to choose otherwise. Second, supposing that a participant chooses to forego immediate economic reward by deliberately responding incorrectly (e.g., out of malice), there are long-term consequences. MTurk maintains an overall ratio of accepted responses to total responses submitted (a "reputation" score), and many requesters only assign work to workers with high ratios (typically around 90%). Since workers cannot easily discard their identities for new ones, incorrect answers have a lasting negative impact on workers. We found that many MTurk workers scrupulously maintain their reputations, sending us e-mails justifying their answers or apologizing for having misunderstood the question.

6. SYSTEM ARCHITECTURE

AUTOMAN is implemented in tiers in order to cleanly separate three concerns: delivering reliable data to the end-user, interfacing with an arbitrary crowdsourcing system, and specifying validation strategies in a crowdsourcing system-agnostic manner. The programmer's interface to AUTOMAN is a domain-specific language embedded in the Scala programming language. The choice of Scala is to maintain full interoperablity with existing Java Virtual Machine code. The DSL abstracts questions at a high level as question functions.

Upon executing a question function, AUTOMAN computes the number of tasks to schedule, the reward, and the timeout; marshals the question to the appropriate backend; and returns immediately, encapsulating work in a Scala `Future`. The runtime memoizes all responses in case the user's program crashes. Once quality control goals are satisfied, AUTOMAN selects and returns an answer.

Each tier in AUTOMAN is abstract and extensible. The default quality control strategy implements the algorithm described in Section 5.2. Programmers can replace the default strategy by implementing the `ValidationStrategy` interface. The default backend is MTurk, but this backend can be replaced with few changes to client code by supplying an `AutomanAdapter` for a different crowdsourcing platform.

7. EVALUATION

We implemented three sample applications using AUTOMAN: a semantic image-classification task using checkboxes (Section 7.1), an image-counting task using radio buttons (Section 7.2), and an optical character recognition (OCR) pipeline using text entry (Section 7.3). These applications were chosen to be representative of the kinds of problems that remain difficult even for state-of-the-art algorithms. We also performed a simulation using real and synthetic traces to explore AUTOMAN's performance as confidence and worker quality is varied (Section 7.4).

7.1. Which item does not belong?

Our first sample application asks users to identify which object does not belong in a collection of items. 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 unnecessary, relatively little code is required to implement this program (27 lines in total).

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 the time of the day. Demographic studies of MTurk have shown that the majority of workers on MTurk are located in the United States and in India.[11] 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.

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.[18] Moreover, vision algorithms that work well for all objects remain elusive.

Counting tasks are trivial with AUTOMAN. We created an image processing pipeline that takes a search string as input, downloads images using Google Image Search, resizes the images, uploads the images to Amazon S3, obscures the URLs using TinyURL, and then posts the question "How many `$items` are in this image?"

We ran this task eight times, spawning 71 question instances at the same time of the day (10 a.m. EST), and employing 861 workers. AUTOMAN ensured that for each of the 71 questions asked, no worker was could participate more than once. Overall, the typical task latency was short. We found that the mean runtime was 8 min, 20 s and that the median runtime was 2 min, 35 s.

The mean is skewed upward by the presence of one long-running task that 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 in order to meet the desired confidence level. AUTOMAN handled this unexpected behavior correctly, running until statistical confidence was reached.

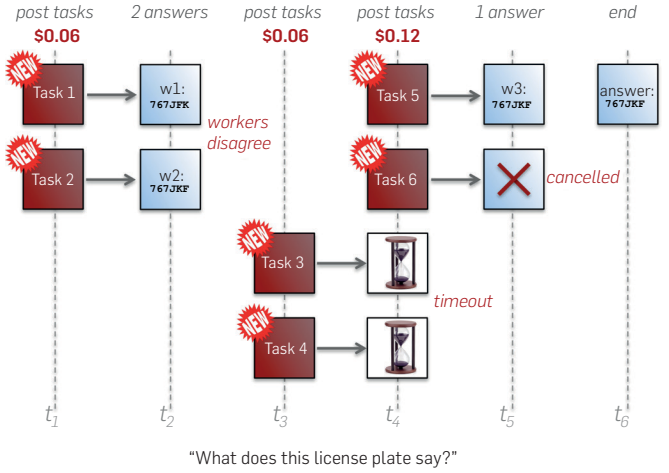
7.3. Automatic license plate recognition

Our last application is the motivating example shown in Figure 1, a program that performs automatic license plate recognition (ALPR). Although ALPR is now widely deployed using distributed networks of traffic cameras, it is still considered a difficult research problem,[8] and academic literature on this subject spans nearly three decades.[5] While state-of-the-art systems can achieve accuracy near 90% under ideal conditions,

[b] The MediaLab LPR database is available at http://www.medialab.ntua.gr/research/LPRdatabase.html.

these systems require substantial engineering in practice.[4] False positives have dramatic negative consequences in unsupervised ALPR systems as tickets are issued to motorists automatically. A natural consequence is that even good unsupervised image-recognition algorithms may need humans in the loop to audit results and to limit false positives.

Figure 3. A sample trace from the ALPR application shown in Figure 1. AutoMan correctly selects the answer `767JKF`, spending a total of $0.18. Incorrect, timed-out, and cancelled tasks are not paid for, saving programmers money.



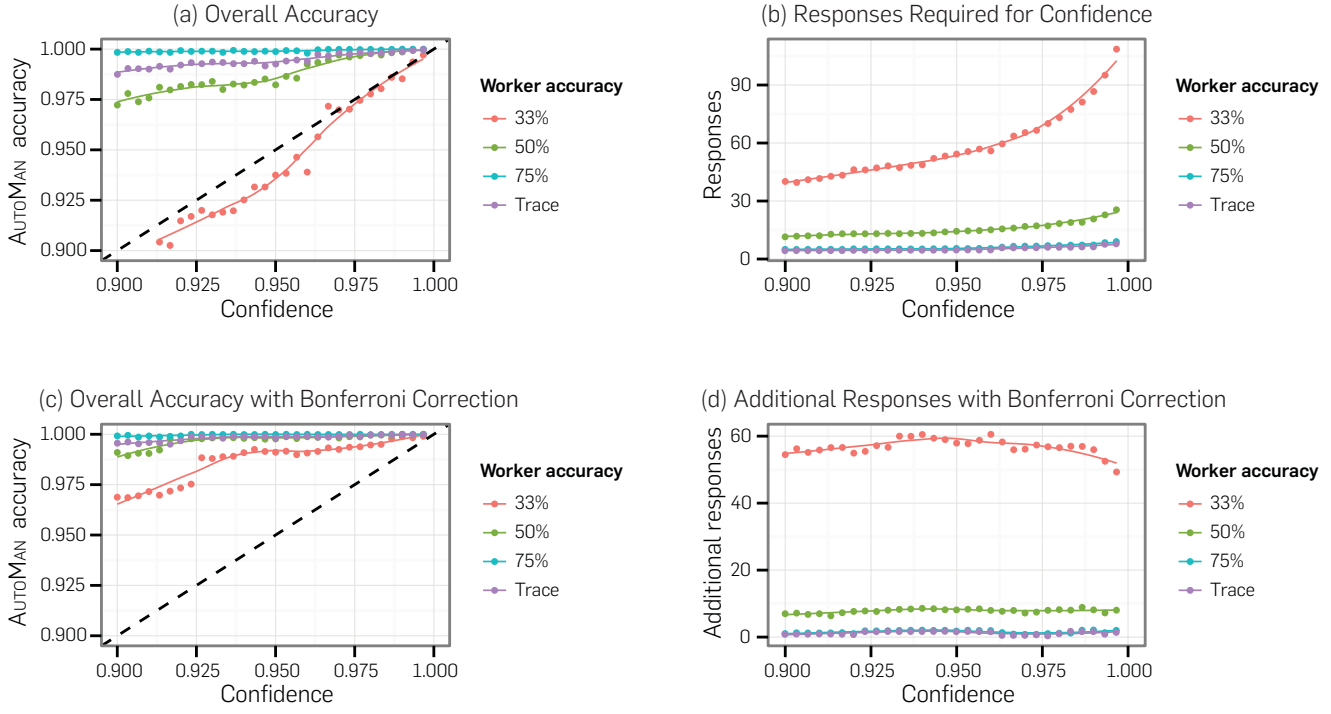
Using AutoMan to engage humans to perform this task required only a few hours of programming time and AutoMan's quality control ensures that it delivers results that match or exceed the state-of-the-art on even the most difficult cases. We evaluated the ALPR application using the MediaLab LPR[b] database. Figure 3 shows a sample trace for a real execution.

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 the "extremely difficult" subset. Each task cost an average of 12.08 cents, with a median latency of less than 2 min per image. AutoMan runs all identification tasks in parallel: one complete run took less than 3 h, while the other took less than 1 h. These translate to throughputs of 24 and 69 plates/h. While the AutoMan application is slower than computer vision approaches, it is simple to implement, and it could be used for only the most difficult images to increase accuracy at low cost.

7.4. Simulation

We simulate AutoMan's ability to meet specified confidence thresholds by varying two parameters, the minimum confidence threshold β, where $0 < \beta < 1$ (we used 50 levels of β), and the probability that a random worker chooses the correct answer $p_r \in \{0.75, 0.50, 0.33\}$. We also simulate worker responses drawn from trace data ("trace") for the "Which item does not belong?" task (Section 7.1). For each setting of

Figure 4. These plots show the effect of worker accuracy on (a) overall accuracy and (b) the number of responses required on a five-option question. "Trace" is a simulation based on real response data while the other simulations model worker accuracies of 33%, 50%, and 75%. Each round of responses ends with a hypothesis test to decide whether to gather more responses, and AutoMan must schedule more rounds to reach the confidence threshold when worker accuracy is low. Naively performing multiple tests creates a risk of accepting a wrong answer, but the Bonferroni correction eliminates this risk by increasing the confidence threshold with each test. Using the correction, AutoMan (c) meets quality control guarantees and (d) requires few additional responses for real workers.



β and p_r we run 10,000 simulations and observe AUTOMAN's response. We classify responses as either *correct* or *incorrect* given the ground truth. *Accuracy* is the mean proportion of correct responses for a given confidence threshold. *Responses required* is the mean number of workers needed to satisfy a given confidence threshold.

Figure 4a and 4b shows accuracy and the number of required responses as a function of β and p_r, respectively. Since the risk of choosing a wrong answer increases as the number of hypothesis tests increases (the "multiple comparisons" problem), we also include figures that show the result of correcting for this effect. Figure 4c shows the accuracy and Figure 4d shows the increase in the number of responses when we apply the Bonferroni bias correction.[10]

These results show that AUTOMAN's quality control algorithm is effective even under pessimistic assumptions about worker quality. AUTOMAN is able to maintain high accuracy in all cases. Applying bias correction ensures that answers meet the programmer's quality threshold even when worker quality is low. This correction can significantly increase the number of additional worker responses required when bad workers dominate. However, worker accuracy tends to be closer to 60%,[20] so the real cost of this correction is low.

8. RELATED WORK

Programming the Crowd. While there has been substantial *ad hoc* use of crowdsourcing platforms, there have been few efforts to manage workers programmatically beyond MTurk's low-level API.

TurKit Script extends JavaScript with a templating feature for common MTurk tasks and adds checkpointing to avoid re-submitting tasks if a script fails.[15] CrowdForge and JabberWocky wrap a MapReduce-like abstraction on MTurk tasks.[1, 13] Unlike AUTOMAN, neither TurKit nor CrowdForge automatically manage scheduling, pricing, or quality control; Jabberwocky uses fixed pricing along with a majority-vote based quality-control scheme.

CrowdDB models crowdsourcing as an extension to SQL for crowdsourcing database cleansing tasks.[9] The query planner tries to minimize the expense of human operations. CrowdDB is not general-purpose and relies on majority voting as its sole quality control mechanism.

Turkomatic crowdsources an entire computation, including the "programming" of the task itself.[14] Turkomatic can be used to construct arbitrarily complex computations, but Turkomatic does not automatically handle budgeting or quality control, and programs cannot be integrated with a conventional programming language.

Quality Control. CrowdFlower is a commercial web service.[17] To enhance quality, CrowdFlower seeds questions with known answers into the task pipeline. CrowdFlower incorporates methods to programmatically generate these "gold" questions to ease the burden on the requester. This approach focuses on establishing trust in particular workers.[12] By contrast, AUTOMAN does not try to estimate worker quality, instead focusing on worker agreement.

Shepherd provides a feedback loop between task requesters and task workers in an effort to increase quality; the idea is to *train* workers to do a particular job well.[7] AUTOMAN requires no feedback between requester and workers.

Soylent crowdsources finding errors, fixing errors, and verifying the fixes.[3] Soylent can handle open-ended questions that AUTOMAN currently does not support. Nonetheless, unlike AUTOMAN, Soylent's approach does not provide any quantitative quality guarantees.

9. CONCLUSION

Humans can perform many tasks with ease that remain difficult or impossible for computers. We present 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 at www.automan-lang.org.

Acknowledgments

This work was supported by the National Science Foundation Grant No. CCF-1144520 and DARPA Award N10AP2026. Andrew McGregor is supported by the National Science Foundation Grant No. CCF-0953754. The authors gratefully acknowledge Mark Corner for his support and initial prompting to explore crowdsourcing.

References

1. Ahmad, S., Battle, A., Malkani, Z., Kamvar, S. The Jabberwocky programming environment for structured social computing. In *UIST 2011*, 53–64.
2. Barowy, D.W, Curtsinger, C., Berger, E.D., McGregor, A. AutoMan: A platform for integrating human-based and digital computation. In *OOPSLA 2012*, 639–654.
3. Bernstein, M.S., Little, G., Miller, R.C., Hartmann, B., Ackerman, M.S., Karger, D.R., Crowell, D., Panovich, K.. Soylent: A word processor with a crowd inside. In *UIST 2010*, 313–322.
4. Chang, G.-L., Zou, N. *ITS Applications in Work Zones to Improve Traffic Operation and Performance Measurements*. Technical Report MD-09-SP708B4G, Maryland Department of Transportation State Highway Administration, May.
5. Davies, P., Emmott, N., Ayland, N. License plate recognition technology for toll violation enforcement. In *IEE Colloquium on Image Analysis for Transport Applications* (Feb 1990), 7/1–7/5.
6. Douceur, J.R.. The Sybil attack. In *IPTPS 2001*, 251–260.
7. Dow, S., Kulkarni, A., Bunge, B., Nguyen, T., Klemmer, S., Hartmann, B. Shepherding the crowd: Managing and providing feedback to crowd workers. In *CHI 2011*, 1669–1674.
8. Due, S., Ibrahim, M., Shehata, M., Badawy, W. Automatic license plate recognition (ALPR): A state of the art review. *IEEE Trans. Circ. Syst. Video Technol. 23* (2012), 311–325.
9. Franklin, M.J., Kossmann, D., Kraska, T., Ramesh, S., Xin, R. CrowdDB: Answering queries with crowdsourcing. In *SIGMOD 2011*, 61–72.
10. Holm, S. A simple sequentially rejective multiple test procedure. *Scand. J. Stat. 6*, 2 (1979), 65–70.
11. Ipeirotis, P.G. *Demographics of Mechanical Turk*. Technical Report CeDER-10-01, NYU Center for Digital Economy Research, 2010.
12. Ipeirotis, P.G., Provost, F., Wang, J. Quality management on Amazon mechanical turk. In *HCOMP 2010*, 64–67.
13. Kittur, A., Smus, B., Khamkar, S., Kraut, R.E. CrowdForge: Crowdsourcing Complex Work.
14. Kulkarni, A.P., Can, M., Hartmann, B. Turkomatic: Automatic recursive task and workflow design for mechanical turk. In *CHI 2011*, 2053–2058.
15. Little, G., Chilton, L.B., Goldman, M., Miller, R.C. TurKit: Human computation algorithms on mechanical turk. In *UIST 2010*, 57–66.
16. Marge, M., Banerjee, S., Rudnicky, A. Using the Amazon mechanical turk for transcription of spoken language. In *ICASSP 2010*, 5270–5273, Mar.
17. Oleson, D., Hester, V., Sorokin, A., Laughlin, G., Le, J., Biewald, L. Programmatic gold: Targeted and scalable quality assurance in crowdsourcing. In *HCOMP 2011*, 43–48.
18. Parikh, D., Zitnick, L. Human-debugging of machines. In *NIPS CSS 2011*.
19. Shahaf, D., Amir, E. Towards a theory of AI completeness. In *Commonsense 2007*.
20. Tamir, D., Kanth, P., Ipeirotis, P. Mechanical turk: Now with 40.92% spam, Dec 2010. www.behind-the-enemy-lines.com.

Daniel W. Barowy, Charlie Curtsinger, Emery D. Berger, and Andrew McGregor ({dbarowy, charlie, emery, mcgregor}@cs.umass.edu), College of Information and Computer Sciences, University of Massachusetts Amherst, 140 Governors Drive, Amherst, MA.