

Soylent: A Word Processor with a Crowd Inside

By Michael S. Bernstein, Greg Little, Robert C. Miller, Björn Hartmann, Mark S. Ackerman, David R. Karger, David Crowell, and Katrina Panovich

Abstract

This paper introduces architectural and interaction patterns for integrating crowdsourced human contributions directly into user interfaces. We focus on writing and editing, complex endeavors that span many levels of conceptual and pragmatic activity. Authoring tools offer help with pragmatics, but for higher-level help, writers commonly turn to other people. We thus present Soylent, a word processing interface that enables writers to call on Mechanical Turk workers to shorten, proofread, and otherwise edit parts of their documents on demand. To improve worker quality, we introduce the Find-Fix-Verify crowd programming pattern, which splits tasks into a series of generation and review stages. Evaluation studies demonstrate the feasibility of crowdsourced editing and investigate questions of reliability, cost, wait time, and work time for edits.

1. INTRODUCTION

Word processing is a complex task that touches on many goals of human-computer interaction. It supports a deep cognitive activity—writing—and requires complicated manipulations. Writing is difficult: even experts routinely make style, grammar, and spelling mistakes. Then, when a writer makes high-level decisions like changing a passage from past to present tense or fleshing out citation sketches into a true references section, she is faced with executing daunting numbers of nontrivial tasks across the entire document. Finally, when the document is a half-page over length, interactive software provides little support to help us trim those last few paragraphs. Good user interfaces aid these tasks; good artificial intelligence helps as well, but it is clear that we have far to go.

In our everyday life, when we need help with complex cognition and manipulation tasks, we often turn to other people. Writing is no exception⁵: we commonly recruit friends and colleagues to help us shape and polish our writing. But we cannot always rely on them: colleagues do not want to proofread every sentence we write, cut a few lines from every paragraph in a 10-page paper, or help us format 30 ACM-style references.

Soylent is a word processing interface that utilizes crowd contributions to aid complex writing tasks ranging from error prevention and paragraph shortening to automation of tasks such as citation searches and tense changes. Using *Soylent* is like having an entire editorial staff available as you write. We hypothesize that crowd workers with a basic knowledge of written English can support both novice and expert writers. These workers perform tasks that the writer might not, such as scrupulously scanning for text to cut or updating a list of addresses to include a zip code. They can

also solve problems that artificial intelligence cannot yet, for example flagging writing errors that the word processor does not catch.

Soylent aids the writing process by integrating paid crowd workers from Amazon's Mechanical Turk platform into Microsoft Word. *Soylent is people*^a: its core algorithms involve calls to Mechanical Turk workers (Turkers). *Soylent* is comprised of three main components:

1. *Shortn*, a text shortening service that cuts selected text down to 85% of its original length on average without changing the meaning of the text or introducing writing errors.
2. *Crowdproof*, a human-powered spelling and grammar checker that finds problems Word misses, explains the error, and suggests fixes.
3. *The Human Macro*, an interface for offloading arbitrary word processing tasks such as formatting citations or finding appropriate figures.

The main contribution of *Soylent* is *the idea of embedding paid crowd workers in an interactive user interface to support complex cognition and manipulation tasks on demand*. This paper contributes the design of one such system, an implementation embedded in Microsoft Word, and a programming pattern that increases the reliability of paid crowd workers on complex tasks. It then expands these contributions with feasibility studies of the performance, cost, and time delay of our three main components and a discussion of the limitations of our approach with respect to privacy, delay, cost, and domain knowledge.

The fundamental technical contribution of this system is a crowd programming pattern called *Find-Fix-Verify*. Mechanical Turk costs money and it can be error-prone; to be worthwhile to the user, we must control costs and ensure correctness. Find-Fix-Verify splits complex crowd intelligence tasks into a series of generation and review stages that utilize independent agreement and voting to produce reliable results. Rather than ask a single crowd worker to read and edit an entire paragraph, for example, Find-Fix-Verify recruits one set of workers to find candidate areas for improvement, another set to suggest improvements to those candidates,

^a With apologies to Charlton Heston (1973): *Soylent is made out of people*.

A full version of this paper was published in *Proceedings of ACM UIST 2010*.

and a final set to filter incorrect candidates. This process prevents errant crowd workers from contributing too much or too little, or introducing errors into the document.

In the rest of this paper, we introduce Soylent and its main components: Shortn, Crowdproof, and The Human Macro. We detail the Find-Fix-Verify pattern that enables Soylent, then evaluate the feasibility of Find-Fix-Verify and our three components.

2. RELATED WORK

Soylent is related to work in two areas: crowdsourcing systems and artificial intelligence for word processing.

2.1. Crowdsourcing

Gathering data to train algorithms is a common use of crowdsourcing. For example, the ESP Game¹⁹ collects descriptions of objects in images for use in object recognition. Mechanical Turk is already used to collect labeled data for machine vision¹⁸ and natural language processing.¹⁷ Soylent tackles problems that are currently infeasible for AI algorithms, even with abundant data. However, Soylent's output may be used to train future AIs.

Soylent builds on work embedding on-demand workforces inside applications and services. For example, Amazon Remembers uses Mechanical Turk to find products that match a photo taken by the user on a phone, and PEST¹⁶ uses Mechanical Turk to vet advertisement recommendations. These systems consist of a single user operation and little or no interaction. Soylent extends this work to more creative, complex tasks where the user can make personalized requests and interact with the returned data by direct manipulation.

Soylent's usage of human computation means that its behavior depends in large part on qualities of crowdsourcing systems and Mechanical Turk in particular. Ross et al. found that Mechanical Turk had two major populations: well-educated, moderate-income Americans, and young, well-educated but less wealthy workers from India.¹⁵ Kittur and Chi⁸ considered how to run user studies on Mechanical Turk, proposing the use of quantitative verifiable questions as a verification mechanism. Find-Fix-Verify builds on this notion of requiring verification to control quality. Heer and Bostock⁶ explored Mechanical Turk as a testbed for graphical perception experiments, finding reliable results when they implemented basic measures like qualification tests. Little et al.¹¹ advocate the use of human computation algorithms on Mechanical Turk. Find-Fix-Verify may be viewed as a new design pattern for human computation algorithms. It is specifically intended to control lazy and overeager Turkers, identify which edits are tied to the same problem, and visualize them in an interface. Quinn and Bederson¹⁴ have authored a survey of human computation systems that expands on this brief review.

2.2. Artificial intelligence for word processing

Soylent is inspired by writers' reliance on friends and colleagues to help shape and polish their writing.⁵

Proofreading is emerging as a common task on Mechanical Turk. Standard Minds^b offers a proofreading service backed by Mechanical Turk that accepts plain text via a web form and returns edits 1 day later. By contrast, Soylent is embedded in a word processor, has much lower latency, and presents the edits in Microsoft Word's user interface. Our work also contributes the Find-Fix-Verify pattern to improve the quality of such proofreading services.

Automatic proofreading has a long history of research⁹ and has seen successful deployment in word processors. However, Microsoft Word's spell checker frequently suffers from false positives, particularly with proper nouns and unusual names. Its grammar checker suffers from the opposite problem: it misses blatant errors.^c Human checkers are currently more reliable, and can also offer suggestions on how to fix the errors they find, which is not always possible for Word—for example, consider the common (but mostly useless) Microsoft Word feedback, "Fragment; consider revising."

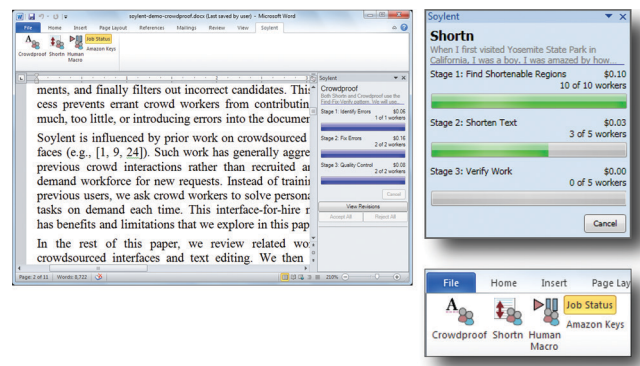
Soylent's text shortening component is related to document summarization, which has also received substantial research attention.¹² Microsoft Word has a summarization feature that uses sentence extraction, which identifies whole sentences to preserve in a passage and deletes the rest, producing substantial shortening but at a great cost in content. Shortn's approach, which can rewrite or cut parts of sentences, is an example of sentence compression, an area of active research that suffers from a lack of training data.³ Soylent's results produce training data to help push this research area forward.

The Human Macro is related to AI techniques for end-user programming. Several systems allow users to demonstrate repetitive editing tasks for automatic execution; examples include Eager, TELS, and Cima.⁴

3. SOYLENT

Soylent is a prototype crowdsourced word processing interface. It is currently built into Microsoft Word (Figure 1), a popular word processor and productivity application.

Figure 1. Soylent adds a set of crowd-powered commands to the word processor.



^b <http://standardminds.com/>.

^c <http://faculty.washington.edu/sandeep/check>.

It demonstrates that computing systems can reach out to crowds to: (1) create new kinds of interactive support for text editing, (2) extend artificial intelligence systems such as style checking, and (3) support natural language commands. These three goals are embedded in Soylent's three main features: text shortening, proofreading, and arbitrary macro tasks.

3.1. Shortn: Text shortening

Shortn aims to demonstrate that crowds can support new kinds of interactions and interactive systems that were very difficult to create before. Some authors struggle to remain within length limits on papers and spend the last hours of the writing process tweaking paragraphs to shave a few lines. This is painful work and a questionable use of the authors' time. Other writers write overly wordy prose and need help editing. Automatic summarization algorithms can identify relevant subsets of text to cut.¹² However, these techniques are less well-suited to small, local language tweaks like those in Shortn, and they cannot guarantee that the resulting text flows well.

Soylent's Shortn interface allows authors to condense sections of text. The user selects the area of text that is too long—for example, a paragraph or section—then presses the Shortn button in Word's Soylent command tab (Figure 1). In response, Soylent launches a series of Mechanical Turk tasks in the background and notifies the user when the text is ready. The user can then launch the Shortn dialog box (Figure 2). On the left is the original paragraph; on the right is the proposed revision. Shortn provides a single slider to allow the user to continuously adjust the length of the paragraph. As the user does so, Shortn computes the combination of crowd trimmings that most closely match the desired length and presents that text to the user on the right. From the user's point of view, as she moves the slider to make the paragraph shorter, sentences are slightly edited, combined

and cut completely to match the length requirement. Areas of text that have been edited or removed are highlighted in red in the visualization. These areas may differ from one slider position to the next.

Shortn typically can remove up to 15%–30% of a paragraph in a single pass, and up to 50% with multiple iterations. It preserves meaning when possible by encouraging crowd workers to focus on wordiness and separately verifying that the rewrite does not change the user's intended meaning. Removing whole arguments or sections is left to the user.

3.2. Crowdproof: Crowdsourced copyediting

Shortn demonstrates that crowds can power new kinds of interactions. We can also involve crowds to augment the artificial intelligence built into applications, for example proofreading. Crowdproof instantiates this idea.

Soylent provides a human-aided spelling, grammar, and style checking interface called Crowdproof (Figure 3). The process finds errors, explains the problem, and offers one to five alternative rewrites. Crowdproof is essentially a distributed proofreader or copyeditor.

To use Crowdproof, the user highlights a section of text and presses the proofreading button in the Soylent ribbon tab. The task is queued to the Soylent status pane and the user is free to keep working. Because Crowdproof costs money, it does not issue requests unless commanded.

When the crowd is finished, Soylent calls out the erroneous sections with a purple dashed underline. If the user clicks on the error, a drop-down menu explains the problem and offers a list of alternatives. By clicking on the desired alternative, the user replaces the incorrect text with an option of his or her choice. If the user hovers over the Error Descriptions menu item, the popout menu suggests additional second-opinions of why the error was called out.

Figure 2. Shortn allows users to adjust the length of a paragraph via a slider. Red text indicates locations where the crowd has provided a rewrite or cut. Tick marks on the slider represent possible lengths.

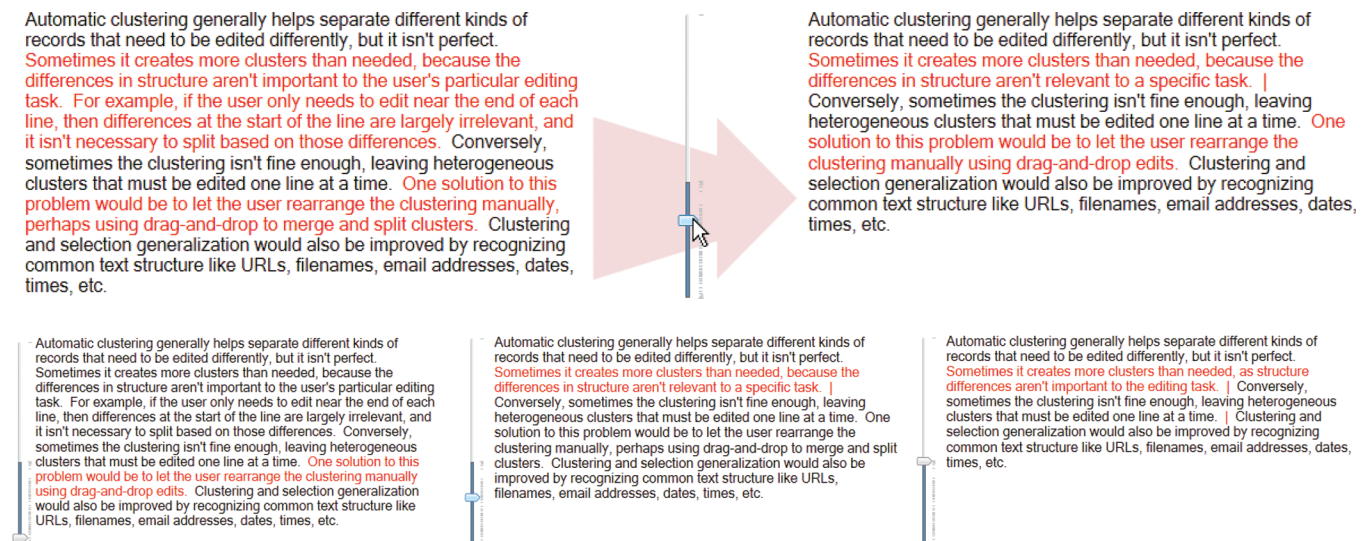
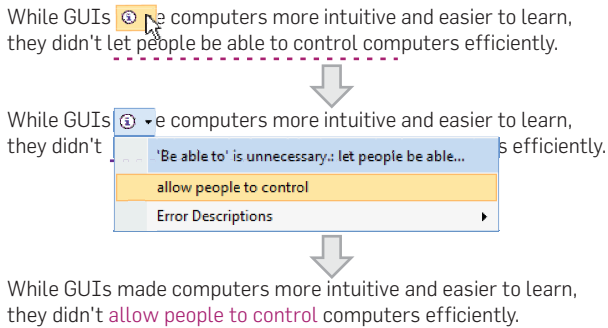


Figure 3. Crowdproof is a human-augmented proofreader. The drop-down explains the problem (blue title) and suggests fixes (gold gold selection).



3.3. The human macro: Natural language crowd scripting

Embedding crowd workers in an interface allows us to reconsider designs for short end-user programming tasks. Typically, users need to translate their intentions into algorithmic thinking explicitly via a scripting language or implicitly through learned activity.⁴ But tasks conveyed to humans can be written in a much more natural way. While natural language command interfaces continue to struggle with unconstrained input over a large search space, humans are good at understanding written instructions.

The Human Macro is Soylent's natural language command interface. Soylent users can use it to request arbitrary work quickly in human language. Launching the Human Macro opens a request form (Figure 4). The design challenge here is to ensure that the user creates tasks that are scoped correctly for a Mechanical Turk worker. We wish to prevent the user from spending money on a buggy command.

The form dialog is split in two mirrored pieces: a task entry form on the left, and a preview of what the crowd worker will see on the right. The preview contextualizes the user's request, reminding the user that they are writing something akin to a Help Wanted or Craigslist advertisement. The form suggests that the user provide an example input and output, which is an effective way to clarify the task requirements to workers. If the user selected text before opening the dialog, they have the option to split the task by each sentence or paragraph, so (for example) the task might be parallelized across all entries on a list. The user then chooses how many separate workers he would like to complete the task. The Human Macro helps debug the task by allowing a test run on one sentence or paragraph.

The user chooses whether the crowd workers' suggestions should replace the existing text or just annotate it. If the user chooses to replace, the Human Macro underlines the text in purple and enables drop-down substitution like the Crowdproof interface. If the user chooses to annotate, the feedback populates comment bubbles anchored on the selected text by utilizing Word's reviewing comments interface.

4. PROGRAMMING WITH CROWDS

This section characterizes the challenges of leveraging crowd labor for open-ended document editing tasks. We introduce

Figure 4. The Human Macro allows users to request arbitrary tasks over their document. Left: user's request pane. Right: crowd worker task preview, which updates as the user edits the request pane.

the Find-Fix-Verify pattern to improve output quality in the face of uncertain crowd worker quality. As we prepared Soylent and explored the Mechanical Turk platform, we performed and documented dozens of experiments.⁴ For this project alone, we have interacted with over 10,000 workers across over 2,500 different tasks. We draw on this experience in the sections to follow.

4.1. Challenges

We are primarily concerned with tasks where crowd workers directly edit a user's data in an open-ended manner. These tasks include shortening, proofreading, and user-requested changes such as address formatting. In our experiments, it is evident that many of the raw results that workers produce on such tasks are unsatisfactory. As a rule-of-thumb, roughly 30% of the results from open-ended tasks are poor. This "30% rule" is supported by the experimental section of this paper as well. Clearly, a 30% error rate is unacceptable to the end user. To address the problem, it is important to understand the nature of unsatisfactory responses.

High variance of effort. Crowd workers exhibit high variance in the amount of effort they invest in a task. We might characterize two useful personas at the ends of the effort spectrum, the *Lazy Turker* and the *Eager Beaver*. The *Lazy Turker* does as little work as necessary to get paid. For example, we asked workers to proofread the following error-filled paragraph from a high school essay site.⁶ Ground-truth errors are colored below, highlighting some of the low quality elements of the writing:

⁴ <http://groups.csail.mit.edu/uid/deneme/>.

⁶ <http://www.essay.org/school/english/ofmicelandmen.txt>.

The theme of loneliness features throughout many scenes in *Of Mice and Men* and is often the dominant theme of sections during this story. This theme occurs during many circumstances but is not present from start to finish. In my mind for a theme to be pervasive is must be present during every element of the story. There are many themes that are present most of the way through such as sacrifice, friendship and comradeship. But in my opinion there is only one theme that is present from beginning to end, this theme is pursuit of dreams.

However, a Lazy Turker inserted only a single character to correct a spelling mistake. The single change is highlighted below:

The theme of loneliness features throughout many scenes in *Of Mice and Men* and is often the dominant theme of sections during this story. This theme occurs during many circumstances but is not present from start to finish. In my mind for a theme to be pervasive is must be present during every element of the story. There are many themes that are present most of the way through such as sacrifice, friendship and comrade-ship. But in my opinion there is only one theme that is present from beginning to end, this theme is pursuit of dreams.

This worker fixed the spelling of the word *comradeship*, leaving many obvious errors in the text. In fact, it is not surprising that the worker chose to make this edit, since it was the only word in the paragraph that would have been underlined in their browser because it was misspelled. A first challenge is thus to discourage workers from exhibiting such behavior.

Equally problematic as Lazy Turkers are Eager Beavers. Eager Beavers go beyond the task requirements in order to be helpful, but create further work for the user in the process. For example, when asked to reword a phrase, one Eager Beaver provided a litany of options:

The theme of loneliness features throughout many scenes in *Of Mice and Men* and is often the principal, significant, primary, preeminent, prevailing, foremost, essential, crucial, vital, critical theme of sections during this story.

In their zeal, this worker rendered the resulting sentence ungrammatical. Eager Beavers may also leave extra comments in the document or reformat paragraphs. It would be problematic to funnel such work back to the user.

Both the Lazy Turker and the Eager Beaver are looking for a way to clearly signal to the requester that they have completed the work. Without clear guidelines, the Lazy Turker will choose the path that produces any signal and the Eager Beaver will produce too many signals.

Crowd workers introduce errors. Crowd workers attempting complex tasks can accidentally introduce substantial new errors. For example, when proofreading paragraphs about the novel *Of Mice and Men*, workers variously changed the title to just *Of Mice*, replaced existing grammar errors with new errors of their own, and changed the text to state that *Of Mice and Men* is a movie rather than a novel. Such errors are compounded if the output of one worker is used as input for other workers.

The result: Low-quality work. These issues compound into what we earlier termed the 30% rule: that roughly one-third of the suggestions we get from workers on Mechanical Turk are not high-enough quality to show an end user. We cannot simply ask workers to help shorten or proofread a paragraph: we need to guide and coordinate their activities.

These two personas are not particular to Mechanical Turk. Whether we are using intrinsic or extrinsic motivators—money, love, fame, or others—there is almost always an un-even distribution of participation. For example, in Wikipedia, there are many Eager Beaver editors who try hard to make edits, but they introduce errors along the way and often have their work reverted.

4.2. The Find-Fix-Verify pattern

Crowd-powered systems must control the efforts of both the Eager Beaver and Lazy Turker and limit the introduction of errors. Absent suitable control techniques for open-ended tasks, the rate of problematic edits is too high to be useful. We feel that the state of programming crowds is analogous to that of UI technology before the introduction of design patterns like Model-View-Controller, which codified best practices. In this section, we propose the Find-Fix-Verify pattern as one method of programming crowds to reliably complete open-ended tasks that directly edit the user's data.^f We describe the pattern and then explain its use in Solyent across tasks like proofreading and text shortening.

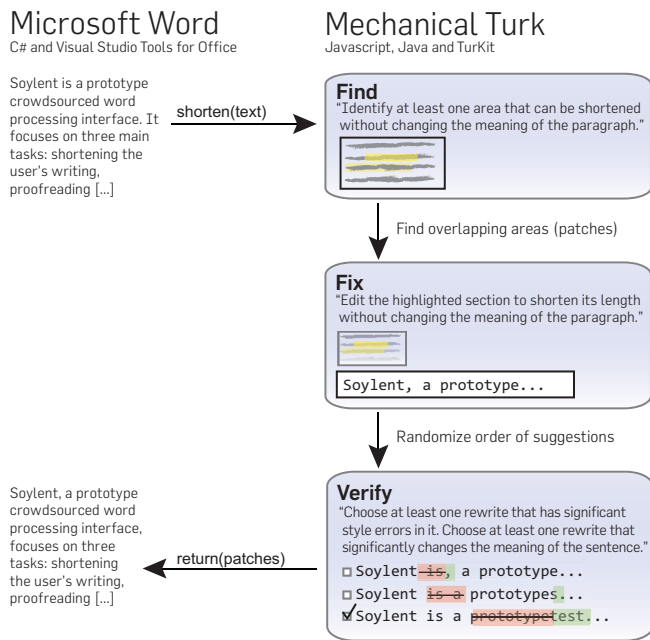
Find-Fix-Verify description. The Find-Fix-Verify pattern separates open-ended tasks into three stages where workers can make clear contributions. The workflow is visualized in Figure 5.

Both Shortn and Crowdproof use the Find-Fix-Verify pattern. We will use Shortn as an illustrative example in this section. To provide the user with near-continuous control of paragraph length, Shortn should produce many alternative rewrites without changing the meaning of the original text or introduce^g grammatical errors. We begin by splitting the input region into paragraphs.

^f Closed-ended tasks like voting can test against labeled examples for quality control.¹⁰ Open-ended tasks have many possible correct answers, so gold standard voting is less useful.

^g Word's grammar checker, eight authors and six reviewers on the original Solyent paper did not catch the error in this sentence. Crowdproof later did, and correctly suggested that "introduce" should be "introducing."

Figure 5. Find-Fix-Verify identifies patches in need of editing, recruits crowd workers to fix the patches, and votes to approve work.



The first stage, Find, asks several crowd workers to identify patches of the user's work that need more attention. For example, when shortening, the Find stage asks ten workers for at least one phrase or sentence that needs to be shortened. Any single worker may produce a noisy result (e.g., Lazy Turkers might prefer errors near the beginning of a paragraph). The Find stage aggregates independent opinions to find the most consistently cited problems: multiple independent agreement is typically a strong signal that a crowd is correct. Soylent keeps patches where at least 20% of the workers agree. These are then fed in parallel into the Fix stage.

The Fix stage recruits workers to revise each agreed-upon patch. Each task now consists of a constrained edit to an area of interest. Workers see the patch highlighted in the paragraph and are asked to fix the problem (e.g., shorten the text). The worker can see the entire paragraph but only edit the sentences containing the patch. A small number (3–5) of workers propose revisions. Even if 30% of work is bad, 3–5 submissions are sufficient to produce viable alternatives. In Shortn, workers also vote on whether the patch can be cut completely. If so, we introduce the empty string as a revision.

The Verify stage performs quality control on revisions. We randomize the order of the unique alternatives generated in the Fix stage and ask 3–5 new workers to vote on them (Figure 5). We either ask workers to vote on the best option (when the interface needs a default choice, like Crowdproof) or to flag poor suggestions (when the interface requires as many options as possible, like Shortn). To ensure that workers cannot vote for their own work, we ban all Fix workers from participating in the Verify stage for that paragraph. To aid comparison, the Mechanical Turk task annotates each rewrite using color and strikethroughs to

highlight its differences from the original. We use majority voting to remove problematic rewrites and to decide if the patch can be removed. At the end of the Verify stage, we have a set of candidate patches and a list of verified rewrites for each patch.

To keep the algorithm responsive, we use a 15-minute timeout at each stage. If a stage times out, we still wait for at least six workers in Find, three workers in Fix, and three workers in Verify.

Pattern discussion. Why should tasks be split into independent Find-Fix-Verify stages? Why not let crowd workers find an error and fix it, for increased efficiency and economy? Lazy Turkers will always choose the easiest error to fix, so combining Find and Fix will result in poor coverage. By splitting Find from Fix, we can direct Lazy Turkers to propose a fix to patches that they might otherwise ignore. Additionally, splitting Find and Fix enables us to merge work completed in parallel. Had each worker edited the entire paragraph, we would not know which edits were trying to fix the same problem. By splitting Find and Fix, we can map edits to patches and produce a much richer user interface—for example, the multiple options in Crowdproof's replacement dropdown.

The Verify stage reduces noise in the returned result. The high-level idea here is that we are placing the workers in productive tension with one another: one set of workers is proposing solutions, and another set is tasked with looking critically at those suggestions. Anecdotally, workers are better at vetting suggestions than they are at producing original work. Independent agreement among Verify workers can help certify an edit as good or bad. Verification trades off time lag with quality: a user who can tolerate more error but needs less time lag might opt not to verify work or use fewer verification workers.

Find-Fix-Verify has downsides. One challenge that the Find-Fix-Verify pattern shares with other Mechanical Turk algorithms is that it can stall when workers are slow to accept the task. Rather than wait for ten workers to complete the Find task before moving on to Fix, a timeout parameter can force our algorithm to advance if a minimum threshold of workers have completed the work. Find-Fix-Verify also makes it difficult for a particularly skilled worker to make large changes: decomposing the task makes it easier to complete for the average worker, but may be more frustrating for experts in the crowd.

5. IMPLEMENTATION

Soylent consists of a front-end application-level add-in to Microsoft Word and a back-end service to run Mechanical Turk tasks (Figure 5). The Microsoft Word plug-in is written using Microsoft Visual Studio Tools for Office (VSTO) and the Windows Presentation Foundation (WPF). Back-end scripts use the TurKit Mechanical Turk toolkit.¹¹

Shortn in particular must choose a set of rewrites when given a candidate slider length. When the user specifies a desired maximum length, Shortn searches for the longest combination of rewrites subject to the length constraint. A simple implementation would exhaustively list all combinations and then cache them, but this approach scales poorly with many patches. If runtime becomes an issue, we can view the search as a multiple-choice knapsack problem. In a

multiple-choice knapsack problem, the items to be placed into the knapsack come from multiple classes, and only one item from each class may be chosen. So, for Shortn, each item class is an area of text with one or more options: each class has one option if it was not selected as a patch, and more options if the crowd called out the text as a patch and wrote alternatives. The multiple-choice knapsack problem can be solved with a polynomial time dynamic programming algorithm.

6. EVALUATION

Our initial evaluation sought to establish evidence for SoyLent’s end-to-end feasibility, as well as to understand the properties of the Find-Fix-Verify design pattern.

6.1. Shortn evaluation

We evaluated Shortn quantitatively by running it on example texts. Our goal was to see how much Shortn could shorten text, as well as its associated cost and time characteristics. We collected five examples of texts that might be sent to Shortn, each between one and seven paragraphs long. We chose these inputs to span from preliminary drafts to finished essays and from easily understood to dense technical material (Table 1).

To simulate a real-world deployment, we ran the algorithms with a timeout enabled and set to 20 minutes for each stage. We required 6–10 workers to complete the Find tasks and 3–5 workers to complete the Fix and Verify tasks: if a Find task failed to recruit even six workers, it might wait

indefinitely. To match going rates on Mechanical Turk, we paid \$0.08 per Find, \$0.05 per Fix, and \$0.04 per Verify.

Each resulting paragraph had many possible variations depending on the number of shortened alternatives that passed the Verify stage—we chose the shortest possible version for analysis and compared its length to the original paragraph. We also measured *wait time*, the time between posting the task and the worker accepting the task, and *work time*, the time between acceptance and submission. In all tasks, it was possible for the algorithm to stall while waiting for workers, having a large effect on averages. Therefore, we report medians, which are more robust to outliers.

Results. Shortn produced revisions that were 78%–90% of the original document length. For reference, a reduction to 85% could slim an 11 $\frac{3}{4}$ page ACM paper draft down to 10 pages with no substantial cuts in the content. Table 1 summarizes and gives examples of Shortn’s behavior. Typically, Shortn focused on unnecessarily wordy phrases like “are going to have to” (Table 1, Blog). Crowd workers merged sentences when patches spanned sentence boundaries (Table 1, Classic UIST Paper), and occasionally cut whole phrases or sentences.

To investigate time characteristics, we separate the notion of wait time from work time. The vast majority of Shortn’s running time is currently spent waiting, because it can take minutes or hours for workers to find and accept the task. Here, our current wait time—summing the median Find, median Fix, and median Verify—was 18.5 minutes (1st Quartile Q_1 = 8.3 minutes, 3rd Quartile Q_3 = 41.6 minutes).

Table 1. Our evaluation run of Shortn produced revisions between 78%–90% of the original paragraph length on a single run.

Input text	Original length	Final length	Work stats	Time per paragraph (min)	Example Output
Blog	3 paragraphs, 12 sentences, 272 words	83% character length	\$4.57, 158 workers	46–57	Print publishers are in a tizzy over Apple’s new iPad because they hope to finally be able to charge for their digital editions. But in order to get people to pay for their magazine and newspaper apps, they are going to have to offer something different that readers cannot get at the newsstand or on the open Web.
Classic UIST Paper ⁷	7 paragraphs, 22 sentences, 478 words	87%	\$7.45, 264 workers	49–84	The metaDESK effort is part of the larger Tangible Bits project: The Tangible Bits vision paper, which introduced the metaDESK along with and two companion platforms, the transBOARD and ambient ROOM.
Draft UIST Paper	5 paragraphs, 23 sentences, 652 words	90%	\$7.47, 284 workers	52–72	In this paper we argue that it is possible and desirable to combine the easy input affordances of text with the powerful retrieval and visualization capabilities of graphical applications. We present WenSo, a tool that which uses lightweight text input to capture richly structured information for later retrieval and navigation in a graphical environment.
Rambling Enron E-mail	6 paragraphs, 24 sentences, 406 words	78%	\$9.72, 362 workers	44–52	A previous board member, Steve Burleigh, created our web site last year and gave me alot of ideas. For this year, I found a web site called eTeamZ that hosts web sites for sports groups. Check out our new page: [...]
Technical Writing ¹	3 paragraphs, 13 sentences, 291 words	82%	\$4.84, 188 workers	132–489	Figure 3 shows the pseudocode that implements this design for Lookup. FAWN-DS extracts two fields from the 160-bit key: the i low order bits of the key (the index bits) and the next 15 low order bits (the key fragment).

The Example Output column contains example edits from each input.

This wait time can be much longer because tasks can stall waiting for workers, as Table 1 shows.

Considering only work time and assuming negligible wait time, Shortn produced cuts within minutes. We again estimate overall work time by examining the median amount of time a worker spent in each stage of the Find-Fix-Verify process. This process reveals that the median shortening took 118 seconds of work time, or just under 2 minutes, when summed across all three stages ($Q_1 = 60$ seconds, $Q_3 = 3.6$ minutes). Using recruitment techniques developed since this research was published, users may see shortening tasks approaching a limit of 2 minutes.

The average paragraph cost \$1.41 to shorten under our pay model. This cost split into \$0.55 to identify an average of two patches, then \$0.48 to generate alternatives and \$0.38 to filter results for each of those patches. Our experience is that paying less slows down the later parts of the process, but it does not impact quality¹³—it would be viable for shortening paragraphs under a loose deadline.

Qualitatively, Shortn was most successful when the input had unnecessary text. For example, with the Blog input, Shortn was able to remove several words and phrases without changing the meaning of the sentence. Workers were able to blend these cuts into the sentence easily. Even the most technical input texts had extraneous phrases, so Shortn was usually able to make at least one small edit of this nature in each paragraph. As Soylent runs, it can collect a large database of these straightforward rewrites, then use them to train a machine learning algorithm to suggest some shortenings automatically.

Shortn occasionally introduced errors into the paragraph. While workers tended to stay away from cutting material they did not understand, they still occasionally flagged such patches. As a result, workers sometimes made edits that were grammatically appropriate but stylistically incorrect. For example, it may be inappropriate to remove the academic signaling phrase “In this paper we argue that...” from an introduction. Cuts were a second source of error: workers in the Fix stage would vote that a patch could be removed entirely from the sentence, but were not given the chance to massage the effect of the cut into the sentence. So, cuts often led to capitalization and punctuation problems at sentence boundaries. Modern auto-correction techniques could catch many of these errors. Parallelism was another source of error: for example, in Technical Writing (Table 1), the two cuts were from two different patches, and thus handled by separate workers. These workers could not predict that their cuts would not match, one cutting the parenthetical and the other cutting the main phrase.

To investigate the extent of these issues, we coded all 126 shortening suggestions as to whether they led to a grammatical error. Of these suggestions, 37 suggestions were ungrammatical, again supporting our rule of thumb that 30% of raw worker edits will be noisy. The Verify step caught 19 of the errors (50% of 37) while also removing 15 grammatical sentences. Its error rate was thus $(18 \text{ false negatives} + 15 \text{ false positives})/137 = 26.1\%$, again near 30%. Microsoft Word’s grammar checker caught 13 of the errors. Combining Word and Shortn caught 24 of the 37 errors.

We experimented with feeding the shortest output from the Blog text back into the algorithm to see if it could continue shortening. It continued to produce cuts between

70%–80% with each iteration. We ceased after 3 iterations, having shortened the text to less than 50% length without sacrificing much by way of readability or major content.

6.2. Crowproof evaluation

To evaluate Crowproof, we obtained a set of five input texts in need of proofreading. These inputs were error-ridden text that passes Word’s grammar checker, text written by an ESL student, quick notes from a presentation, a poorly written Wikipedia article (Dandu Monara), and a draft UIST paper. We manually labeled all spelling, grammatical and style errors in each of the five inputs, identifying a total of 49 errors. We then ran Crowproof on the inputs using a 20-minute stage timeout, with prices \$0.06 for Find, \$0.08 for Fix, and \$0.04 for Verify. We measured the errors that Crowproof caught, that Crowproof fixed, and that Word caught. We ruled that Crowproof had caught an error if one of the identified patches contained the error.

Results. Soylent’s proofreading algorithm caught 33 of the 49 errors (67%). For comparison, Microsoft Word’s grammar checker found 15 errors (30%). Combined, Word and Soylent flagged 40 errors (82%). Word and Soylent tended to identify different errors, rather than both focusing on the easy and obvious mistakes. This result lends more support to Crowproof’s approach: it can focus on errors that automatic proofreaders have not already identified.

Crowproof was effective at fixing errors that it found. Using the Verify stage to choose the best textual replacement, Soylent fixed 29 of the 33 errors it flagged (88%). To investigate the impact of the Verify stage, we labeled each unique correction that workers suggested as grammatical or not. Fully 28 of 62 suggestions, or 45%, were ungrammatical. The fact that such noisy suggestions produced correct replacements again suggests that workers are much better at verification than they are at authoring.

Crowproof’s most common problem was missing a minor error that was in the same patch as a more egregious error. The four errors that Crowproof failed to fix were all contained in patches with at least one other error; Lazy Turkers fixed only the most noticeable problem. A second problem was a lack of domain knowledge: in the ESL input, workers did not know what a GUI was, so they could not know that the author intended “GUIs” instead of “GUI.” There were also stylistic opinions that the original author might not have agreed with: in the Draft UIST input, the author had a penchant for triple dashes that the workers did not appreciate.

Crowproof shared many running time characteristics with Shortn. Its median work time was 2.8 minutes ($Q_1 = 1.7$ minutes, $Q_3 = 4.7$ minutes), so it completes in very little work time. Similarly to Shortn, its wait time was 18 minutes (Median = 17.6, $Q_1 = 9.8$, $Q_3 = 30.8$). It cost more money to run per paragraph ($\mu = \$3.40$, $\sigma = \$2.13$) because it identified far more patches per paragraph: we chose paragraphs in dire need of proofreading.

6.3. Human macro evaluation

We were interested in understanding whether end users could instruct Mechanical Turk workers to perform open-ended tasks. Can users communicate their intention clearly? Can workers execute the amateur-authored tasks correctly?

Method. We generated five feasible Human Macro scenarios. These scenarios included changing the tense of a story from past tense to present tense, finding a Creative Commons-licensed image to illustrate a paragraph, giving feedback on a draft blog post, gathering BibTeX for some citations, and filling out mailing addresses in a list. We recruited two sets of users: five undergraduate and graduate students in our computer science department (4 males) and five administrative associates in our department (all females). We showed each user one of the five prompts, consisting of an example input and output pair. We purposefully did not describe the task to the participants so that we would not influence how they wrote their task descriptions. We then introduced participants to The Human Macro and described what it would do. We asked them to write a task description for their prompt using The Human Macro. We then sent the description to Mechanical Turk and requested that five workers complete each request. In addition to the ten requests generated by our participants, one author generated five requests himself to simulate a user who is familiar with Mechanical Turk.

We coded results using two quality metrics: intention (did the worker understand the prompt and make a good faith effort?) and accuracy (was the result flawless?). If the worker completed the task but made a small error, the result was coded as good intention and poor accuracy.

Results. Users were generally successful at communicating their intention. The average command saw an 88% intention success rate (max = 100%, min = 60%). Typical intention errors occurred when the prompt contained two requirements: for example, the Figure task asked both for an image and proof that the image is Creative Commons-licensed. Workers read far enough to understand that they needed to find a picture, found one, and left. Successful users clearly signaled Creative Commons status in the title field of their request.

With accuracy, we again see that roughly 30% of work contained an error. (The average accuracy was 70.8%.) Workers commonly got the task mostly correct, but failed on some detail. For example, in the Tense task, some workers changed all but one of the verbs to present tense, and in the List Processing task, sometimes a field would not be correctly capitalized or an Eager Beaver would add too much extra information. These kinds of errors would be dangerous to expose to the user, because the user might likewise not realize that there is a small error in the work.

7. DISCUSSION

This section reviews some fundamental questions about the nature of paid, crowd-powered interfaces as embodied in SoyLent. Our work suggests that it may be possible to transition from an era where Wizard of Oz techniques were used only as prototyping tools to an era where a “Wizard of Turk” can be permanently wired into a system. We touch on resulting issues of wait time, cost, legal ownership, privacy, and domain knowledge.

In our vision of interface outsourcing, authors have immediate access to a pool of human expertise. Lag times in our current implementation are still on the order of minutes to hours, due to worker demographics, worker availability, the relative attractiveness of our tasks, and so on. While future growth in crowdsourced work will likely shorten lag

times, this is an important avenue of future work. It may be possible to explicitly engineer for responsiveness in return for higher monetary investment, or to keep workers around with other tasks until needed.²

With respect to cost, SoyLent requires that authors pay all workers for document editing—even if many changes never find their way into the final work product. One might therefore argue that interface outsourcing is too expensive to be practical. We counter that in fact all current document processing tasks also incur significant cost (in terms of computing infrastructure, time, software and salaries); the only difference is that interface outsourcing precisely quantifies the price of each small unit of work. While payment-per-edit may restrict deployment to commercial contexts, it remains an open question whether the gains in productivity for the author are justified by the expense.

Regarding privacy, SoyLent exposes the author’s document to third party workers without knowing the workers’ identities. Authors and their employers may not want such exposure if the document’s content is confidential or otherwise sensitive. One solution is to restrict the set of workers that can perform tasks: for example, large companies could maintain internal worker pools. Rather than a binary opposition, a continuum of privacy and exposure options exists.


SoyLent also raises questions over legal ownership of the resulting text, which is part-user and part-Turker generated. Do the Turkers who participate in Find-Fix-Verify gain any legal rights to the document? We believe not: the Mechanical Turk worker contract explicitly states that it is work-for-hire, so results belong to the requester. Likewise with historical precedent: traditional copyeditors do not own their edits to an article. However, crowd-sourced interfaces will need to consider legal questions carefully.

It is important that the research community ask how crowd-sourcing can be a social good, rather than a tool that reinforces inequality. Crowdsourcing is a sort of renewal of scientific management. Taylorism had positive impacts on optimizing workflows, but it was also associated with the dehumanizing elements of factory work and the industrial revolution. Similarly, naive crowdsourcing might treat people as a new kind of abstracted API call, ignoring the essential humanness of these sociotechnical systems. Instead, we need to evolve our design process for crowdsourcing systems to involve the crowds workers’ perspective directly.

8. CONCLUSION

The following conclusion was Shortn’ed to 85% length: This chapter presents SoyLent, a word processing interface that uses crowd workers to help with proofreading, document shortening, editing and commenting tasks. SoyLent is an example of a new kind of interactive user interface in which the end user has direct access to a crowd of workers for assistance with tasks that require human attention and common sense. Implementing these kinds of interfaces requires new software programming patterns for interface software, since crowds behave differently than computer systems. We have introduced one important pattern, Find-Fix-Verify, which splits complex editing tasks into a series of identification, generation, and verification stages that use independent agreement and voting produce reliable results. We evaluated SoyLent with a range of editing tasks, finding and correcting 82% of grammar

errors when combined with automatic checking, shortening text to approximately 85% of original length per iteration, and executing a variety of human macros successfully.

Future work falls in three categories. First are new crowd-driven features for word processing, such as readability analysis, smart find-and-replace (so that renaming “Michael” to “Michelle” also changes “he” to “she”) and figure or citation number checking. Second are new techniques for optimizing crowd-programmed algorithms to reduce wait time and cost. Finally, we believe that our research points the way toward integrating on-demand crowd work into other authoring interfaces, particularly in creative domains like image editing and programming. 

References

- Andersen, D.G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., Vasudevan, V. FAWN: A fast array of wimpy nodes. In *Proc. SOSP '09* (2009).
- Bigham, J.P., Jayant, C., Ji, H., Little, G., Miller, A., Miller, R.C., Miller, R., Tatrowicz, A., White, B., White, S., Yeh, T. VizWiz: Nearly real-time answers to visual questions. In *Proc. UIST '10* (2010).
- Clarke, J., Lapata, M. Models for sentence compression: A comparison across domains, training requirements and evaluation measures. In *Proc. ACL '06* (2006).
- Cypher, A. *Watch What I Do*. MIT Press, Cambridge, MA, 1993.
- Dourish, P., Bellotti, V. Awareness and coordination in shared workspaces. In *Proc. CSCW '92* (1992).
- Heer, J., Bostock, M. Crowdsourcing graphical perception: Using Mechanical Turk to assess visualization design. In *Proc. CHI '10* (2010).
- Ishii, H., Ullmer, B. Tangible bits: Towards seamless interfaces between people, bits and atoms. In *Proc. UIST '97* (1997).
- Kittur, A., Chi, E.H., Suh, B. Crowdsourcing user studies with Mechanical Turk. In *Proc. CHI '08* (2008).
- Kukich, K. Techniques for automatically correcting words in text. *ACM Comput. Surv. (CSUR)* 24, 4 (1992), 377–439.
- Le, J., Edmonds, A., Hester, V., Biewald, L. Ensuring quality in crowdsourced search relevance evaluation: The effects of training question distribution. In *Proc. SIGIR 2010 Workshop on Crowdsourcing for Search Evaluation* (2010), 21–26.
- Little, G., Chilton, L., Goldman, M., Miller, R.C. TurkKit: Human computation algorithms on Mechanical Turk. In *Proc. UIST '10* (2010) ACM Press.
- Marcu, D. *The Theory and Practice of Discourse Parsing and Summarization*. MIT Press, Cambridge, MA, 2000.
- Mason, W., Watts, D.J. Financial incentives and the performance of crowds. In *Proc. HCOMP '09* (2009) ACM Press.
- Quinn, A.J., Bederson, B.B. Human computation: A survey and taxonomy of a growing field. In *Proc. CHI '11* (2011) ACM.
- Ross, J., Irani, L., Silberman, M.S., Zaldivar, A., Tomlinson, B. Who are the crowdworkers? Shifting demographics in Amazon Mechanical Turk. In *alt.chi '10* (2010) ACM Press.
- Sala, M., Partridge, K., Jacobson, L., Begole, J. An exploration into activity-informed physical advertising using PEST. In *Pervasive '07*, volume 4480 of *Lecture Notes in Computer Science* (Berlin, Heidelberg, 2007), Springer, Berlin, Heidelberg.
- Snow, R., O'Connor, B., Jurafsky, D., Ng, A.Y. Cheap and fast—But is it good? Evaluating non-expert annotations for natural language tasks. In *Proc. ACL '08* (2008).
- Sorokin, A., Forsyth, D. Utility data annotation with Amazon Mechanical Turk. *Proc. CVPR '08* (2008).
- von Ahn, L., Dabbish, L. Labeling images with a computer game. In *CHI '04* (2004).

Michael S. Bernstein (msb@cs.stanford.edu) Stanford University, Stanford, CA.

Robert C. Miller (rcm@mit.edu) MIT CSAIL, Cambridge, MA.

Björn Hartmann (bjoern@cs.berkeley.edu) Computer Science Division University of California, Berkeley, CA.

David R. Karger (karger@mit.edu) MIT CSAIL, Cambridge, MA.

Greg Little (glittle@csail.mit.edu)

David Crowell (dcrowell.mit@gmail.com)

Mark S. Ackerman (ackerm@umich.edu) Computer Science & Engineering University of Michigan, Ann Arbor, MI.

Katrina Panovich (kpan@google.com), Google, Inc., Mountain View, CA.

© 2015 ACM 0001-0782/15/08 \$15.00



Watch the authors discuss their work in this exclusive *Communications* video.
<http://cacm.acm.org/videos/soylent-a-word-processor-with-a-crowd-inside>

World-Renowned Journals from ACM

ACM publishes over 50 magazines and journals that cover an array of established as well as emerging areas of the computing field. IT professionals worldwide depend on ACM's publications to keep them abreast of the latest technological developments and industry news in a timely, comprehensive manner of the highest quality and integrity. For a complete listing of ACM's leading magazines & journals, including our renowned Transaction Series, please visit the ACM publications homepage: www.acm.org/pubs.

ACM Transactions on Interactive Intelligent Systems



ACM Transactions on Interactive Intelligent Systems (TIIS). This quarterly journal publishes papers on research encompassing the design, realization, or evaluation of interactive systems incorporating some form of machine intelligence.

ACM Transactions on Computation Theory



ACM Transactions on Computation Theory (ToCT). This quarterly peer-reviewed journal has an emphasis on computational complexity, foundations of cryptography and other computation-based topics in theoretical computer science.

PLEASE CONTACT ACM MEMBER SERVICES TO PLACE AN ORDER

Phone: 1.800.342.6626 (U.S. and Canada)
 +1.212.626.0500 (Global)

Fax: +1.212.944.1318
 (Hours: 8:30am–4:30pm, Eastern Time)

Email: acmhelp@acm.org

Mail: ACM Member Services
 General Post Office
 PO Box 30777
 New York, NY 10087-0777 USA



Association for
Computing Machinery

Advancing Computing as a Science & Profession

www.acm.org/pubs