# Tutorial-based Interfaces for Cloud-enabled Applications

**Gierad Laput, Eytan Adar**
School of Information
University of Michigan
Ann Arbor, MI 48109
{glaput,eadar}@umich.edu

**Mira Dontcheva, Wilmot Li**
Advanced Technology Labs
Adobe
San Francisco, CA 94103
{mirad,wilmotli}@adobe.com


Figure 1: TAPPCLOUD turns static online tutorials into *tutorial-based applications* (tapps). Once a tapp has been created, a TAPPCLOUD bookmarklet appears on the source tutorial page (a). Clicking on the bookmarklet opens the TAPPCLOUD wiki that hosts all created tapps (b). Users can upload their own images (c) to a tapp to apply the target technique (d).

## ABSTRACT

Powerful image editing software like Adobe Photoshop and GIMP have complex interfaces that can be hard to master. To help users perform image editing tasks, we introduce *tutorial-based applications* (tapps) that retain the step-by-step structure and descriptive text of tutorials but can also automatically apply tutorial steps to new images. Thus, tapps can be used to batch process many images automatically, similar to traditional macros. Tapps also support interactive exploration of parameters, automatic variations, and direct manipulation (e.g., selection, brushing). Another key feature of tapps is that they execute on remote instances of Photoshop, which allows users to edit their images on any Web-enabled device. We demonstrate a working prototype system called TAPPCLOUD for creating, managing and using tapps. Initial user feedback indicates support for both the interactive features of tapps and their ability to automate image editing. We conclude with a discussion of approaches and challenges of pushing monolithic direct-manipulation GUIs to the cloud.

**ACM Classification:** H.5.2 [Information interfaces and presentation]: User Interfaces. - Graphical user interfaces.

**Keywords:** tutorials, macros, cloud computing.


*UIST'12,* October 7-10, 2012, Cambridge, MA, USA.


## INTRODUCTION

Advances in camera technology have made it easier than ever to capture high quality images. As a result, photo editing, which has traditionally been the domain of experienced photographers and creative professionals, has become a common task for the general population, many of whom want to edit their own digital photos to improve image quality, refine composition, or create specific stylized effects. Tools such as the Instagram and Hipstamatic mobile applications offer simple one-button image manipulation solutions. However these applications are limited to reproducing a single specific effect that may not correspond to what the user wants to do. At the other extreme are full image-editing packages such as Adobe Photoshop and GIMP. While these tools are powerful (with hundreds of different features and parameters), they are also difficult to learn and seem like overkill for many of the relatively simple image-editing tasks that users often want to perform (e.g., red-eye removal, cropping, simple filters).

Tutorials and application-specific macros offer two potential solutions. It is easy to find a wide variety of step-by-step tutorials that explain how to accomplish many different photo editing tasks in Photoshop. Unfortunately, such tutorials can be hard to follow, especially for inexperienced users who may not have the necessary prior knowledge or familiarity with the user interface to understand the steps. Even, if the user succeeds in following the tutorial, he still has to manually apply the technique to every image he wants to edit. On the other hand, macros allow users to automatically apply a technique to their own images without actually performing the steps in the target application. However, a typical macro

does not expose any of the underlying details of the image editing technique that it encodes, which makes it difficult for the user to explore the space of possible effects that the technique can generate or adapt the technique to different images. In most cases, the macro simply acts as a black-box that takes an image as input and produces a modified image as output (e.g., transforming a photograph into a watercolor picture). Moreover, using a macro requires the presence of the target application. Applications such as Adobe Photoshop require fairly powerful hardware and high resolution displays to execute effectively, which makes it impossible to run macros on cameras and other mobile devices.

In this work, we introduce *tutorial-based applications* (tapps) that address some of the limitations of tutorials and macros. A tapp is a specialized application that encodes a target technique as an ordered set of commands that are executed on a traditional monolithic GUI application (possibly hosted elsewhere). In the context of Adobe Photoshop, tapps enable users to apply image editing techniques described in online Adobe Photoshop tutorials directly to their own images. Tapps have several important features:

**Generated from tutorials.** Our system provides semi-automated tools that help users generate tapps from input tutorials by mapping tutorial steps to Photoshop commands. Once a tapp is created, it can be shared with other users.

**Step-by-step structure.** Instead of applying Photoshop commands as a black box macro, a tapp retains the step-by-step structure and descriptive text of the original tutorial and presents this structure as an interface for applying the target technique. This approach allows users to see and understand the sequence of manipulations that are applied to new input images.

**Interactive steps.** When executing a tapp, users can customize the technique for different images by interactively modifying the parameters of individual steps. As the user changes a parameter setting, the tapp interface shows real-time updates to the manipulated image. This design allows users to apply tapps in a flexible manner without having to familiarize themselves with the entire application UI.

**Automatic variations.** To help users explore the design space of the target technique, we support parameter variations in the interface (e.g., the radius on a blur filter). Variations can be pre-calculated and displayed in a "gallery" mode next to the step or explored interactively. These variants are also pushed through what remains of the tutorial to demonstrate the effects of the setting on all later steps. Because TAPPCLOUD can use many instances of Photoshop running in parallel, variants can be calculated as quickly as a single execution of the tutorial.

**Separate interface from application.** By separating the tapp interface from the original application we are able to push the application itself into the cloud, which allows a user to access many instances of an application from a single interface. The difficultly we begin to address in this paper is how to take a monolithic, direct-manipulation GUI and make it function effectively as a cloud-based application. We start by eliminating as much direct-manipulation interactions as possible but add new techniques to support direct-manipulation of parallel versions of the same application. The advantage of this architecture is that it allows users to run tapps on any device that is connected to the Internet, even if it does not have Photoshop installed. Furthermore, connecting to multiple Photoshop instances enables tapps to work with multiple images or generate many variations in a responsive manner.

We demonstrate these features in our TAPPCLOUD system, which consists of the TAPPCLOUD server and a collection of application servers (see Figure 2). The TAPPCLOUD server includes three modules: the *tutorial parser* helps users convert existing tutorials to tapps by automatically extracting application commands and parameter values from tutorial text; for commands that cannot be extracted automatically, the *programming-by-demonstration (PBD) mapper* allows users to interactively associate tutorial steps with commands and expose adjustable parameters by demonstrating them in the application; finally, the TAPPCLOUD *wiki* hosts the converted tapps. The application servers, which are hosted in the cloud, run instances of the target application that execute commands when a user runs a tapp.

Our work makes several contributions:

- We present an approach for transforming tutorials into applications that leverages both automated analysis and demonstration interfaces. We demonstrate a few forms of these applications including a wiki-style interface and file browser extensions.
- We propose interaction techniques for making tutorials act as applications (parameter setting, mixed access when direct manipulation is necessary).
- Finally, we introduce a framework for pushing desktop-based applications to the cloud, including strategies for keeping the application servers synchronized and interaction techniques for working with multiple application instances.

## RELATED WORK

Our system design is inspired by work in three main areas: instructional content analysis, authoring instructional content with demonstrations, and virtual computing.

### Instructional content analysis

Interest in improving Web search and use of online help resources has spurred a number of research efforts in analyzing webpages to extract deeper structure, including application commands. One class of approaches uses a list of all possible application commands [6, 8, 24] to search learning documents for exact or approximate string matches. Unfortunately, many application commands use words that are common in the English language, such as "image," "select," or "copy," which leads to errors. Lau et. al. [18] take a more principled approach and compare three approaches for extracting commands from written instructions for Web-based tasks, such as creating an email account or buying a product. They compare a keyword-based interpreter, a grammar-based interpreter, and a machine learning-based interpreter and find that the machine learning approach is best in recognizing

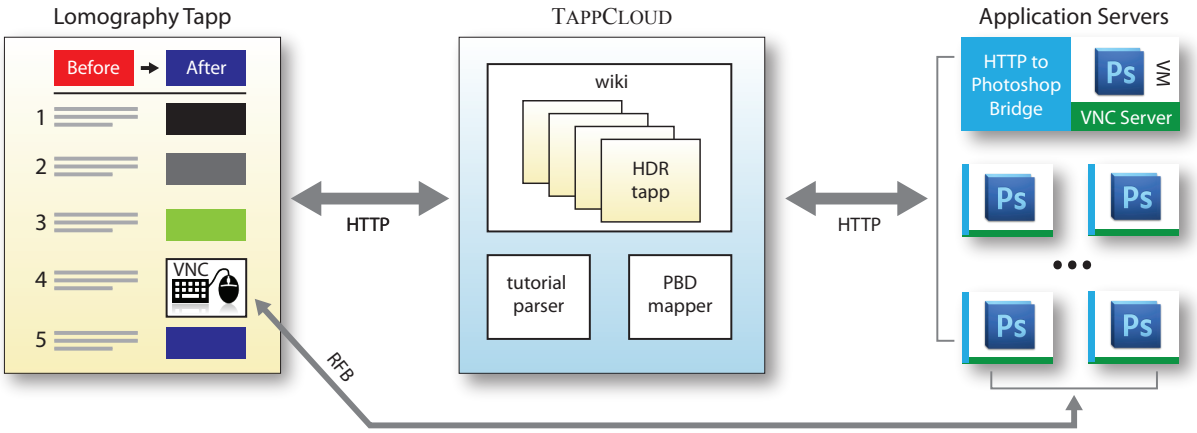


Figure 2: The TAPPCLOUD architecture. The main server (center) serves individual tapps to the user. Here it is serving a tapp that generates Lomo-style images (left). The server also acts to manage the creation and modification of tapps. The application servers (right) run in the cloud and interact with the server or with the user through our parallel-VNC system.

commands in written instruction (82% accuracy) and the keyword-based interpreter is best at interpreting an instruction (59% accuracy). Interpreting an instruction is inherently more difficult because it requires putting together multiple pieces of information. For example, in the instruction "Create a gaussian blur with $Filter > Blur > Gaussian\ Blur$ and change the radius to 10 pixels," it is easier to identify the gaussian blur command as a valid command than it is to interpret that the radius is a parameter of the command and that it should be set to 10. In recent work Fourney et al. [7] show that it is possible to recognize commands from step-by-step desktop application tutorials with up to 95% accuracy. We build on these approaches and use a Conditional Random Field (CRF) extractor augmented with heuristics. Our text-analysis approach performs on par with previous approaches though we identify a larger number of entity classes (menu commands, tools, parameters, etc.).

### Tutorials and macros from demonstrations

Even in the limited domain of instructional content, natural language understanding is a challenging problem. Thus, some researchers have developed systems for producing instructional content from example demonstrations [3, 19, 9]. Tutorials that are produced by demonstration are generally easier to author but also have the added benefit of being highly structured. Thus, they can easily be embedded in applications and offer step-by-step instruction as the user needs it. TAPPCLOUD makes use of demonstrations to enhance tutorial analysis. When our tutorial parser fails to interpret a command, we let users give TAPPCLOUD a demonstration of how to accomplish the task. We present both automatically extracted and demonstrated commands as text using text templates. This approach was inspired by Grabler et al.'s system for authoring tutorials for image-editing applications from demonstrations [9].

In many ways TAPPCLOUD tutorials are closer in spirit to application macros than to traditional tutorials, as they are not explicitly trying to teach users how to use an application. They are simply enabling users to accomplish a particular task while leveraging the familiar paradigm of tutorials. Creating generalizable macros in the context of visual editing was first introduced by the Chimera system [13], which used heuristics to generalize macros for manipulating 2D graphics. More recently Berthouzoz et al. [4] presents a framework and interface for content-adaptive macros for image editing. Their macros automatically adapt parameter settings, selections, and brush strokes based on the user's image. TAPPCLOUD does not currently adapt tutorial settings automatically, but such functionality would be a nice addition to the existing system. As TAPPCLOUD tutorials interact with multiple application instances, users can explore the parameter design space for one or more images in parallel. Interacting with variations has been explored in the context of specific interface elements [28]. TAPPCLOUD builds on this work and allows users to explore variations across an entire workflow.

### Virtual computing

There are several commercial Virtual Network Computing (VNC) systems that offer the ability to access remote desktop applications. Commercial vendors have also begun to port their desktop applications to the Web (e.g., Photoshop Express [25]) though these interfaces are generally much more limited than their desktop counterparts. Additionally, these systems are intended to support one-to-one (one user to one application) or collaborative (multiple users to one application) interactions. In contrast, TAPPCLOUD is designed to allow single users to interact with multiple application instances simultaneously. Sophisticated systems such as Façade [26], Prefab [5], Sikuli [32], and WinCuts [27] allow users to crop and manipulate GUIs both in local and remote settings but are similarly intended for single-user or single-application domains.

Another line of research focuses on enabling what is normally possible in a desktop browser on a mobile phone (e.g., Highlight [20]). These systems allow users to run macros authored by demonstration remotely on a server that is running a full browser. While we build on the ideas introduced by these systems, the direct-manipulation aspect of our image editing cloud-enabled applications requires the development of new interaction techniques.


Figure 3: Authoring a tapp. To convert an existing online tutorial (top left), the user first applies our tutorial parser to automatically link commands to tutorial text (a). Next, he manually links unparsed steps and selects adjustable parameters to expose using our PBD mapper (b). The final tapp is saved to the TAPPCLOUD wiki (c).

## TAPP WORKFLOW

We begin by describing how users author tapps and apply them to their own images with TAPPCLOUD. The next section provides details on the system implementation.

### Tapp Authoring Scenario

Let's follow Mario as he uses TAPPCLOUD to create a new tapp from an online "Orton effect" tutorial. Mario opens the tutorial page and clicks the "Convert to Tapp" bookmarklet. TAPPCLOUD generates a new tapp wiki page. All of the steps where the system is able to extract the corresponding application command are highlighted (Steps 4 and 5 in Figure 3a). For all other steps, Mario manually specifies the appropriate command by demonstration. Here, he selects the relevant command text in Step 6 ("Multiply for blend mode") and performs the action in Photoshop, as shown in Figure 3b. Based on this demonstration, the TAPPCLOUD Programming-by-Demonstration (PBD) mapper automatically links the command with the step. Once a step is associated with an executable command, Mario can expose adjustable parameters by selecting the relevant text and indicating that it represents a parameter value. For example, if the tutorial parser failed to identify the parameter in Step 5, Mario can select the "2" (the blur radius), and TAPPCLOUD binds this text to the corresponding command parameter. If the command has more than one adjustable parameter, TAPPCLOUD lets the user select the appropriate one from a list. Once Mario is done linking steps to commands and exposing adjustable parameters, he saves the tapp to the wiki (Figure 3c).

### Tapp Use Scenario

Let's follow Julie as she uses TAPPCLOUD to edit some of her photos. Julie finds a tutorial describing a "Lomo effect" and wants to apply it to one of her images. She clicks the "TAPPCLOUD" bookmarklet (Figure 1a), which takes her to the TAPPCLOUD wiki where she sees the "Lomo" tapp that corresponds to that webpage (Figure 1b). Julie clicks on the "Upload your own image" button and selects her image. Julie sees the steps of the tutorial applied to her own photo (Figure 1d) with intermediate images visualizing the effect of each step. Julie changes the opacity parameter for the black fill layer to create more contrast (Figure 4a). She clicks on "View as Gallery," which shows her the effect of the layer opacity at a glance (Figure 4b). She then clicks "Apply to the rest of the steps" and sees a grid of images that describe the effect of the layer opacity on the intermediate and final images. Julie likes the way the middle column looks, so she double clicks the middle image to choose this setting. The TAPPCLOUD wiki saves all of Julie's selected settings so that she can access them at any time.

Some tapps include steps that require direct interaction with the canvas. For example, when Julie wants to remove a stranger from a vacation photo, she opens the "Remove object" tapp, which asks her to select the region of the image to remove. She clicks on "Demonstrate" to open a live view editing window and selects the relevant image region (Figure 5). When she is done, the rest of the steps are applied to her selection. To explore the parameter space of a direct manipulation step, Julie can use a parallel live view mode that allows her to perform a single interaction with several different parameter settings. For example, Julie can specify different brush parameters for a step that involves painting on the canvas. When she paints, our system dispatches the mouse actions to multiple Photoshop instances, each of which uses a different brush setting. Julie can switch between the live view windows to see how the parameters affect the results.

Finally, Julie can apply tapps that do not require canvas interactions to multiple images in a batch processing mode. For instance, Julie can upload several images via the TAPPCLOUD wiki. Similarly, she can apply a tapp to local images on her computer through a custom context menu item. Julie can click on a folder with the right mouse button to open the context menu and select the menu item that invokes the tapp. The images in the folder are pushed to the TAPPCLOUD server, and when the processing is complete the results are saved in Julie's original folder.

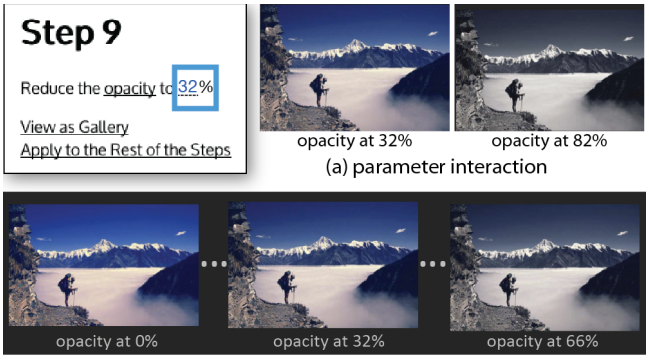


Figure 4: Interacting with parameters. Within a tapp, users can directly modify parameter settings and see the results of the change (a) or view galleries of images that show a range of results across a variety of parameter values (b).

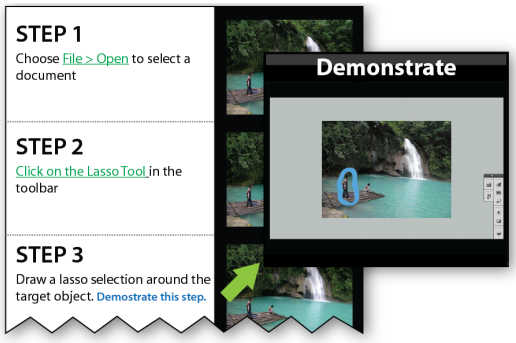


Figure 5: Live editing. TAPPCLOUD provides a live editing interface that allows users to perform steps interactively. Here, the user selects a specific region of the image to remove.

## THE TAPPCLOUD SYSTEM

We now describe the TAPPCLOUD system in more detail.

### Tutorial Parser

Ideally, tutorial text would function as a form of literate programming [11]. However, because of the extensive direct manipulation directives issued by the authors of the tutorials (e.g., "paint a few more smoke lines using the brush tool") and the highly informal writing (e.g., "Anyway, on top of the other layers just add the Invert adjustment") it is extremely difficult to transform an arbitrary tutorial into a program that can be directly executed. Nonetheless, there are many steps that can be extracted automatically by our parser. The parser generates a "skeleton" program for a tapp that can then be augmented, modified, or extended through the tapp construction interface.

In order to automatically detect instructions and parameters in tutorial text found on the Web, we implemented a Conditional Random Field (CRF) extractor augmented with some heuristics. The complete set of features for this extractor are described in Appendix 1.

Our working dataset included 630 tutorials pulled from two websites: psd.tutsplus.com (549 tutorials) and psdbox.com (81 tutorials). Because of the regular structure of tutorials on these sites, each tutorial was automatically deconstructed into individual steps (14,345 in total). A training set was constructed using 400 tutorials (748,780 labeled tokens including 1,762 MENU entities, 3,018 TOOL entities, and 12,189 PARAMETERS). The remaining 230 tutorials were used as a test collection (722 MENUS, 1,221 TOOLS, and 4,677 PARAMETERS). The CRF extractor attained a precision/recall/F1 of .95/.99/.97 for menus, .97/.98/.98 for tools, and .67/.25/.36 for parameters.

In order to associate parameter values with the parameter names, a simple heuristic creates an association between the nearest numerical value found in the text to the parameter name (without passing over "boundary" signals such as semicolons, commas, the token 'and,' or periods). Once identified, commands and parameters are associated to the tutorial step where they are located, and a programmatic "stub" in ExtendScript (a JavaScript variant that is understood by Adobe applications) is generated. In a test set of 4,482 steps, we were able to extract entities (tools, menus, etc.) in 2,072 of them (46%). While we believe this number may be somewhat improved, a completely unsupervised system would be difficult to achieve. Because of this, we extend our system with support for programming by demonstration and direct modification to these skeleton scripts.

### Programming-by-Demonstration (PBD) Mapper

For commands that are not detected by the tutorial parser, we provide a demonstration interface for associating commands with tutorial text. The tapp author selects the relevant text, opens Photoshop, and then performs the appropriate action. The PBD Mapper automatically captures the relevant ExtendScript code for the demonstrated command (using Photoshop's built-in ScriptListener plugin) and associates it with the selected text.

While a tapp author may demonstrate a specific step to be included in the program, we also allow the user to demonstrate all of the steps in the tutorial at one time. We convert the trace created by the demonstration into a step-by-step tutorial using the technique described in [9] and [14] and map it to the pre-processed tapp skeleton. To map the executable commands output from the demonstration to the partial extractions obtained by the CRF or other annotation process, we compare each step in the user's trace to the existing tapp stubs. Commands in the trace for which there is a match can be automatically copied into the tapp. For those commands that cannot be matched, we rely on the fact that both traces and tutorials are roughly linear. An unmatched trace command(s) is frequently book-ended by commands that do match. This allows for an approximate association between the commands and tapp steps.

### Running tutorials as applications

When users want to use a tapp on their own images, they can access TAPPCLOUD through a webpage from the TAPPCLOUD wiki (see Figure 2) or directly from their file system through a JSON API [10]. The TAPPCLOUD server selects an available application server and sends each user's image along with the list of programmatic stubs that correspond to the tapp steps. Because of our focus on Photoshop, TAPPCLOUD uses ExtendScript. We note that currently TAP-

PCLOUD can only support applications that have a scripting interface. Many complex, monolithic GUIs (e.g., Photoshop, Illustrator, Word, Excel, etc.) offer scripting capabilities. Alternatively, automation systems external to the application may also be used (e.g., AutoIt [2], Sikuli [32], or Prefab [5]).

TAPPCLOUD talks to multiple application servers to distribute calculations required to execute the tapp. For a tapp handling batch image requests, for example, TAPPCLOUD can map each image to a different application server. Our TAPPCLOUD implementation provides a simple FIFO queue when the number of requests exceed the number of available servers. Clearly, other complicated schemes are possible and we briefly suggest an example in the Discussion Section.

Although the wiki provides a convenient front-end for users, because we use a JSON API, there are many other possibilities for interaction. For example, we have implemented a custom menu item for the desktop. We created a Python shell extension [23] which calls the TAPPCLOUD server using a remote URL request. We scan for images within the folder and use them as inputs for the tapp request. Once processed, the results are saved back to the original folder.

### Tapp interaction

*Real-time parameter modification* For commands that are bound to specific parameters, TAPPCLOUD provides a level of interaction which allows a user to see the immediate effect of a given value. We use the Tangle library [30] for binding tutorial text to corresponding controls for parameter adjustment (e.g., sliders and text input). Users can adjust parameter-bound elements by clicking on the parameter and dragging the mouse to the left or right to increase or decrease the parameter value. The effect of this change is displayed automatically within the current and subsequent steps.

*Exploring parameter ranges* TAPPCLOUD also provides a gallery mode to support the exploration of parameter values. The gallery mode provides a side-by-side comparison of the effect of a parameter value. When enabled, a row of images along with the parameter value used in the step are displayed (see Figure 4). We further support exploration by allowing the user to see the effects of parameter values as they propagate throughout the rest of the steps via an image gallery "grid." Users can explore the gallery and immediately see how a given value can affect the overall result. By clicking on a particular image in the gallery, the tapp is set to those parameter values and saved for subsequent steps.

*Rich interactions through VNC* When direct manipulation of graphical content is necessary, TAPPCLOUD provides a limited "window" to the remote application by means of a Virtual Network Computing (VNC) client. VNC uses the Remote Framebuffer (RFB) protocol to send bitmap images of the remote server to a local client and to convey interactions (mouse movements, key presses, etc.) from the client to the server. Because of the relative simplicity of the protocol, many clients have been built including ones that run natively in the browser (using the HTML5 canvas), as applets, or as independent applications. For our purposes, we utilize a modified version of TightVNC [29] that we execute as an applet.

When a direct-manipulation interaction or demonstration is necessary, an embedded applet window allows the user to directly control the remote Photoshop instance. If the user needs to demonstrate a complex set of steps, he can access the entire desktop of the remote instance. However, we believe that a more common situation will require quick interactions with the drawing surface alone. Because of this, we have written a small program to calculate the coordinates of the Photoshop window and modified the VNC client to crop to that region. By programmatically pre-selecting the appropriate tool for the user, they can control the drawing surface to accomplish a task before moving on to the next step (e.g., "draw a mask around the hair.").

A more critical limitation of VNC is that it is intended to pair one client to one server. Instead of requiring users to perform their actions on each instance serially–as the VNC architecture would require–we modified TightVNC to support simultaneous connections to multiple VNC servers. The user has access to any of the drawing surfaces of the servers they are connected to. However, any user interaction is dispatched to all servers by "broadcasting" the RFB protocol packet. Users may "disentangle" some portion of the servers (manipulate one set one way, and another set another) or choose to interact with only one at a time. We have also implemented a feature to support divergent editing of any text dialog (e.g., specifying the feathering on a selection tool or color on a paintbrush). A user currently does this by opening a dialog in our VNC client and entering a comma-delimited list of values. Each of these values is sent to a different server (e.g., a GUI version of a "Map" step in a Map/Reduce operation). The user may now continue to perform parallel manipulation, but potentially may be drawing with a differently configured tool–allowing them to quickly test different ideas in parallel (e.g., recoloring the sky in different shades, smudging at different levels, or sampling differently in a clone stamp).

### Experimental Platform

To test TAPPCLOUD in a realistic environment we deployed our system on Amazon's EC2 cloud running 8 machines (4 small and 4 medium instances running Windows). Small instances are configured with 1 core, 1.7GB RAM, and 160 GB storage whereas a medium instance is 2 cores, 3.75GB RAM, and 410 GB in storage. All instances are running Microsoft Windows Server 2008 with both Photoshop CS5 and a beta version of Photoshop CS6 installed. The cost to operate an instance is $\$.115$ and $\$.230$ per hour for the small and medium instances respectively (at a cost of roughly $\$1000$ per year per machine if running continuously). To construct our instances, one initial machine was configured and then cloned to ensure identical set up.

In addition to Photoshop, each machine also runs a VNC server and a small Java-based Web service that converts HTTP requests to the Photoshop remote connection protocol. This architecture allows TAPPCLOUD to pass ExtendScript commands over the Web and collect thumbnail snapshots of the current image. The main TAPPCLOUD server (Figure 2) is built in Ruby on Rails. We note that our implementation works as effectively on the local desktop with direct-manipulation steps performed by simply switching to

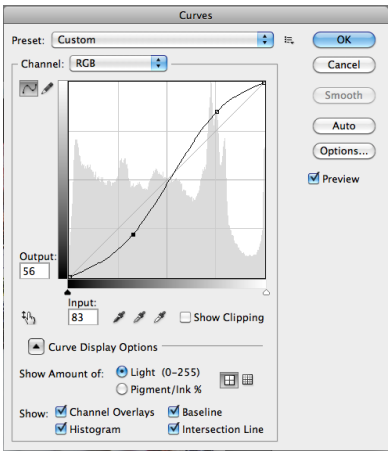


Figure 6: The curves image adjustment allows users to manipulate contrast with a tone curve. TAPPCLOUD doesn't currently support interaction with this type of interface element. To edit an image with this command, the user has to go into "live view" mode and directly interact with Photoshop.

the locally-running Photoshop (though at a loss of features such as cropping to the drawing area). Much of our early testing was done in this configuration.

## DISCUSSION

As we developed TAPPCLOUD, and on receiving informal feedback, we were able to identify potential issues in cloud-enabling applications. A number of these issues were fixed through various design features (described throughout this paper). However, we have also identified other areas for future development.

### Awareness

Designers of direct-manipulation systems build features into their systems to provide feedback even when the system is busy. At the simplest, progress bars or busy icons indicate that the system is "thinking." More complex solutions, such as displaying partial results, also indicate to the user what is happening (or more critically, that something is happening). Within the context of TAPPCLOUD, this is a much more difficult proposition. A tapp may include multiple automated steps when only limited feedback is provided. During this time, the system may be queuing a job, processing–multiple, computationally intensive steps–at the instance nodes, or transferring data. Awareness of what is going on, how much work is left to do, if the system has crashed, and so on, is crucial and more work is necessary to decide how best to convey this.

A second form of "awareness" in the context of parallel instances is how a user tracks divergent workflows and maintains an understanding of how the work is split up and which instance is running which image with which parameters. Others have worked on graphical representations to show high-level system state (i.e., trees or directed acyclic graphs) or interaction techniques such as Side Views [28]. Currently, we provide this information through the gallery view, which is a basic grid representation. This is appealing as it is easy to understand, but does not capture all possible workflow paths.

### User Interface

We feel that we have only begun to explore how to enable users to explore the design space of an image editing technique. Our current interface allows users to manipulate parameters one step at a time. But parameters and steps can interact and to truly understand their interaction, the user must be able explore parameter variations in parallel. Of course even a small number of parameters can quickly lead to combinatorial explosion of possible paths, so we have to carefully consider how to enable such an interface. Berthouzoz et al. [4] present a framework for creating content-adaptive macros from multiple examples. Automatically adapting parameter settings based on the user's image will be very useful for those who want a one button solution, but modeling the relationship between images and user operations could also be useful in building an exploratory interface.

Another future direction is improving the step-by-step tutorial presentation. By experimenting with different tutorials, we found that tutorials often include steps that don't affect the canvas. Some tutorial steps, such as manipulating layers, describe how to manipulate the user interface, not how to edit the image. In future work we plan to group steps that do not have a visible effect. Automatically collapsing the text for such steps may be hard to do fully automatically. Our wiki approach enables users to modify the text and remove unnecessary text or add rationale for a specific step. Our current system doesn't support tying multiple application commands to one step, but extending the system with this support is straightforward. What is less clear is how to provide a rich text editor for the tapp, while still maintaining a mapping between the text and application commands.

Unfortunately, tutorials don't offer a natural interface for all image editing commands. For example, to adjust contrast in an image, photographers often use the *Curves Adjustment* (see Figure 6). In handwritten tutorials, authors often include screenshots showing how to adjust the curves, but applying the curves adjustment often requires image-specific changes. Furthermore, it is unclear how to parameterize this type of control to allow users to easily explore variations. The space of all curves is equivalent to the space of all images.

### Live Demonstration

Our implementation uses a web-based VNC client to perform commands on the remote application server. An alternative to the VNC strategy–which we do not implement in this version of the system–is to create a thin "client" that uses a locally-managed drawing surface (e.g., the canvas or an application such as Deviant Art's Muro [1]) that acts as a proxy to the remote instance. This may work in situations where a selection can be encoded and transmitted programmatically (e.g., specifying a path, converting that to the appropriate ExtendScript command, and pushing the program to the remote instance). This has a number of benefits in situations when security needs to be maintained or a network connection is slow. Conversely, the solution adds to implementation complexity that can grow with the expanding features of the underlying GUI.

### Synchronization

An issue unaddressed in our current system is synchronization between multiple application instances. The virtual ma-

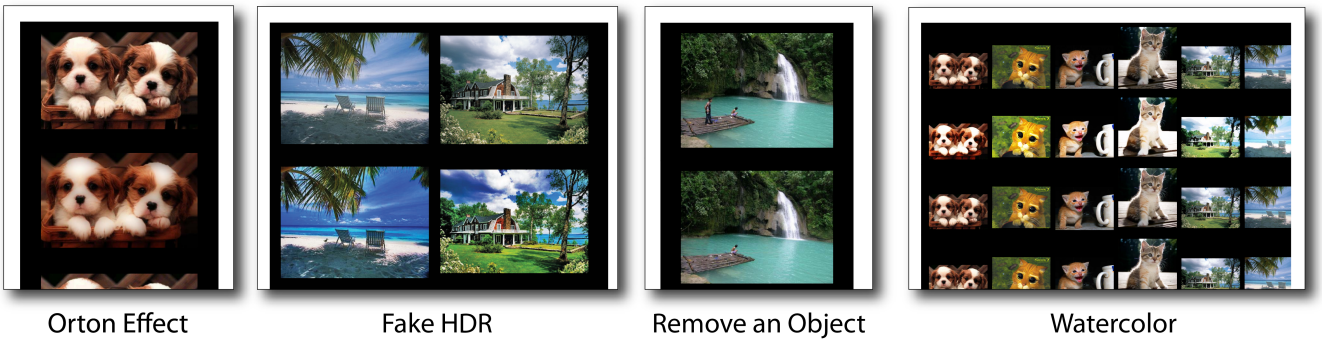


Figure 7: Before-After results of example tapps ranging from Macro-like (Orton and Fake HDR) to those that require direct manipulation (Object removal) and a multi-image processing (Watercolor).

chines running each instance may become out of sync for a number of reasons ranging from low-level issues (different machine specifications, different latencies to the machines, reboots, etc.) to application issues (divergent parameters on an operation taking a different amount of time, the GUIs placed in incompatible states, errors on a machine, etc.). There are two fundamental issues that need to be resolved: detection of unintentional divergence of state, and reconciling unsynchronized VMs. Both problems are well beyond the scope of this paper, though notably there are a number of new VM technologies that support synchronization–generally by logging VM behavior, checkpointing, and transferring over the deltas [16]. Such systems can also support bringing up a new VM set to the state of a currently executing one (live replication) which would allow for dynamic, demand-based VM creation (presently we pre-instantiate a cluster). A similar, high-level, solution would be to checkpoint at the application level to support "resetting" to a known state.

A second synchronization issue is how to reconcile instances that have been made explicitly divergent in response to parameter variants introduced by the user. In this case, a user creates divergent instances (e.g., trying color or font variants) and then wants to pick a dominant one to continue from. In part, this is an awareness problem (how has a tapp diverged and converged), but technically might be addressed through similar techniques to those described above: checkpointing, synchronization, and live-replication.

Lastly, our TAPPCLOUD implementation uses a FIFO queue when the number of processing requests exceed the total number of application servers. Clearly, more sophisticated schemes are possible. For example, when parameter variation occurs in only one step of the entire tapp, overall throughput may be optimized by using a single application server that reverts the step with the variation and re-executes it with a new parameter value rather than sending the complete tapp to multiple application servers to execute the entire tapp.

### Security

In addition to the replication issue, there are a number of security concerns that will need to be addressed if TAPPCLOUD or other "cloudified" GUIs are deployed. Some security complications may be mitigated by creating instances that can only be accessed by a single user at any one time (rather than a shared instance) and encrypting communication. The addition of this security will by necessity create additional overhead, but would be necessary for a live deployment.

## USER FEEDBACK

To solicit user feedback, we carried out two sets of evaluations with Photoshop users (n=13, 5 female/8 male, 25-45 years old). Our goals were 1) to gauge users' response to using tutorial-based applications, and 2) to understand whether users would be interested in exploring the design space of a technique by scrubbing parameter values and using galleries. In our initial evaluation, we demonstrated several tapps to eight users and had an informal discussion about the concept. In our second evaluation, we gathered feedback by asking five users to interact with the TAPPCLOUD wiki, upload photos and edit them with various tapps.

We found consistent positive feedback about the idea of tutorial-based applications. All of the participants expressed that tapps provide an efficient way to get results quickly. The majority of the interviewees agreed that tapps let them skip the workflow details and focus on the result. One user said, "it's really great to just load an image and see it executed in action." One user was particularly positive about the idea of the TAPPCLOUD wiki, and exclaimed that, "the draw of contributing to the wiki and gaining benefits from other tapps is particularly strong." All users said that they would see themselves using tapps in a variety of ways. One particular user said "I want it for mobile!"

Most users agreed that text descriptions were necessary to understand certain steps, especially those that require them to complete direct manipulation steps (e.g., selecting a specific object). Not surprisingly, some users interested in learning were concerned that tapps might conflict with the learning benefits associated with manually going through a tutorial. In fact, some users felt the text was not necessary or too verbose for some steps. Tutorial-based applications are not meant to replace tutorials and could be used in parallel. For example, tapps might offer a debugging interface for someone who gets stuck going through the steps of tutorials. In addition, tutorial text is still useful for automated command extraction and to indicate what commands the user must perform using the demonstration interface.

With regards to the exploration interfaces, users were generally receptive about the instant feedback when changing specific parameters. One particular user said that being able

to easily explore the design space was beneficial, as long as enough context about the parameter was available to make better sense of the effect. Another user felt overwhelmed by the gallery mode. He said, "I usually have a narrow browser window, so this might be too much." More research is needed to establish the best interfaces for exploring the design space of a workflow.

Overall, all interviewed users expressed positive views about the core idea of tutorial-based applications, and they consistently stated that TAPPCLOUD provides a convenient image editing interface.

## CONCLUSIONS

We present an approach and a working implementation for tutorial-based applications that provide web interfaces to traditional desktop applications running in the cloud. We demonstrate the feasibility of this approach with a complex image-editing application, Adobe Photoshop. Early user feedback confirms that enabling users to embed their images in tutorials has benefits and argues for further research in interfaces for exploring the design space of editing workflows.

We have yet to explore the social aspect of tutorial-based applications. The TAPPCLOUD wiki faces some of the same challenges as the AdaptableGIMP [15] and CoScripter [19], including how to fail gracefully, how to manage customizations, and how to surface relevant content. Kong et al. [12] present possible interfaces for our wiki, while Lau et al. [17] suggest reusing repositories of macros to enable speech interfaces. In the future, we hope to identify other mechanisms for invoking and interacting with tapps on different devices and through different modalities (e.g., mobile) that can benefit from the availability of cloud-hosted interfaces.

## ACKNOWLEDGEMENTS

We would like to thank our study participants and the reviewers for their time and feedback. We thank Jason Linder for his help with the figures and Ryan Oswald for early work on the CRF system.

## REFERENCES

1. Deviant Art. Muro. `http://muro.deviantart.com/`. Online; accessed April 10, 2012.
2. AutoIt. AutoIt/autoItScript home. `http://www.autoitscript.com`. Online; accessed April 10, 2012.
3. Lawrence Bergman, Vittorio Castelli, Tessa Lau, and Daniel Oblinger. Docwizards: a system for authoring follow-me documentation wizards. In *Proceedings of UIST*, pages 191–200. ACM, 2005.
4. Floraine Berthouzoz, Wilmot Li, Mira Dontcheva, and Maneesh Agrawala. A framework for content-adaptive photo manipulation macros: Application to face, landscape, and global manipulations. *ACM Trans. Graph.*, 30(5):120:1–120:14, October 2011.
5. Morgan Dixon and James Fogarty. Prefab: implementing advanced behaviors using pixel-based reverse engineering of interface structure. In *Proceedings of SIGCHI*, pages 1525–1534. ACM, 2010.
6. Michael Ekstrand, Wei Li, Tovi Grossman, Justin Matejka, and George Fitzmaurice. Searching for software learning resources using application context. In *Proceedings of UIST*, pages 195–204. ACM, 2011.
7. Adam Fourney, Ben Lafreniere, Richard Mann, and Michael Terry. "then click ok!" extracting references to interface elements in online documentation. In *Proceedings of the SIGCHI*. ACM, 2012.
8. Adam Fourney, Richard Mann, and Michael Terry. Query-feature graphs: bridging user vocabulary and system functionality. In *Proceedings of UIST*, pages 207–216. ACM, 2011.
9. Floraine Grabler, Maneesh Agrawala, Wilmot Li, Mira Dontcheva, and Takeo Igarashi. Generating photo manipulation tutorials by demonstration. *ACM Trans. Graph.*, 28(3):66:1–66:9, July 2009.
10. JSON. JavaScript Object Notation. `http://www.json.org`. Online; accessed April 10, 2012.
11. D.E. Knuth and Stanford University. Computer Science Dept. *Literate programming*. Center for the Study of Language and Information, 1992.
12. Nicholas Kong, Tovi Grossman, Björn Hartmann, George Fitzmaurice, and Maneesh Agrawala. Delta: A tool for representing and comparing workflows. In *Proceedings of the SIGCHI*. ACM, 2012.
13. David Kurlander and Steven Feiner. A history-based macro by example system. In *Proceedings of UIST*, pages 99–106. ACM, 1992.
14. Adobe Labs. Tutorial builder. `http://labs.adobe.com/technologies/tutorialbuilder/`. Online; accessed June 25, 2012.
15. Benjamin Lafreniere, Andrea Bunt, Matthew Lount, Filip Krynicki, and Michael A. Terry. AdaptableGIMP: designing a socially-adaptable interface. In *Adjunct proceedings of UIST*, pages 89–90. ACM, 2011.
16. H.A. Lagar-Cavilla, J.A. Whitney, A.M. Scannell, P. Patchin, S.M. Rumble, E. De Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12. ACM, 2009.
17. Tessa Lau, Julian Cerruti, Guillermo Manzato, Mateo Bengualid, Jeffrey P. Bigham, and Jeffrey Nichols. A conversational interface to web automation. In *Proceedings of UIST*, pages 229–238. ACM, 2010.
18. Tessa Lau, Clemens Drews, and Jeffrey Nichols. Interpreting written how-to instructions. In *Proceedings of IJCAI*, pages 1433–1438. Morgan Kaufmann Publishers Inc., 2009.

19. Gilly Leshed, Eben M. Haber, Tara Matthews, and Tessa Lau. Coscripter: automating & sharing how-to knowledge in the enterprise. In *Proceedings of SIGCHI*, pages 1719–1728. ACM, 2008.
20. Jeffrey Nichols, Zhigang Hua, and John Barton. Highlight: a system for creating and deploying mobile web applications. In *Proceedings of UIST*, pages 249–258. ACM, 2008.
21. Naoaki Okazaki. CRFsuite: a fast implementation of conditional random fields (CRFs), 2007.
22. Hoifung Poon and Pedro Domingos. Joint inference in information extraction. In *Proceedings of AAAI*, pages 913–918. AAAI Press, 2007.
23. Mark Hammond. Pywin32 - Python for Windows Extensions `http://starship.python.net/crew/mhammond/win32/`. Online; accessed July 3, 2012.
24. Vidya Ramesh, Charlie Hsu, Maneesh Agrawala, and Björn Hartmann. Showmehow: translating user interface instructions between applications. In *Proceedings of UIST*, pages 127–134. ACM, 2011.
25. Adobe Software. Photoshop Express. `http://www.photoshop.com/tools/expresseditor`. Online; accessed April 10, 2012.
26. Wolfgang Stuerzlinger, Olivier Chapuis, Dusty Phillips, and Nicolas Roussel. User interface façades: towards fully adaptable user interfaces. In *Proceedings of UIST*, pages 309–318. ACM, 2006.
27. Desney S. Tan, Brian Meyers, and Mary Czerwinski. Wincuts: manipulating arbitrary window regions for more effective use of screen space. In *CHI '04 extended abstracts on Human factors in computing systems*, pages 1525–1528. ACM, 2004.
28. Michael Terry and Elizabeth D. Mynatt. Side views: Persistent, on-demand previews for open-ended tasks. In *Proceedings of UIST*, pages 71–80. ACM Press, 2002.
29. TightVNC. TightVNC: Remote control / remote desktop software. `http://www.tightvnc.com`. Online; accessed April 10, 2012.
30. Bret Victor. tangle: explorable explanations made easy. `http://worrydream.com/#!/Tangle`. Online; accessed April 10, 2012.
31. Wavii. Pretty Fast Parser (pfp). `https://github.com/wavii/pfp`. Online; accessed June 25, 2012.
32. Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. Sikuli: using gui screenshots for search and automation. In *Proceedings of UIST*, pages 183–192. ACM, 2009.

| *Feature Description* | *Example* |
|---|---|
| First token in sentence | *Hello* world |
| Last token in sentence | Hello *world* |
| numeric | 99 ; 93.5 |
| greater-than symbol | > |
| single character | j ; I ; z |
| capitalized | Gradient |
| punctuation | . ; ? |
| unit | px ; pixels ; inches |
| percent | % |
| command key | cmd ; ctrl ; alt |
| token itself | Holding |
| normalized token | hold |
| Part of Speech (POS) of token [31] | VB ; IN ; DT |
| previous/next tokens (window size 1 to 3) | |
| normalized previous/next tokens | |
| POS of previous/next tokens | |

Table 1: CRF Features

**APPENDIX 1: CRF Extractor**

To preform command extraction from unstructured text we make use of the CRFsuite package [21]. We utilize the features described in Table 1.

To create a training set, we utilized a self-supervised training technique that automatically identified high-quality matches for menus, parameters, and tools in the text. This was done by forcing a direct text match to menu commands that was generated semi-automatically (a total of 1,168 menus, panel names, tool names, etc. for Photoshop). For example, the command list might include the menu command "$Filter > Blur > Lens\ Blur$" which brings up a dialog with the parameters *noise*, *radius*, etc. To generate a training set, all menu and tool commands found in our data collection were marked as matches (e.g., every instance of "$Filter > Blur > Lens\ Blur$" was marked as a match to the MENU class and all matches to the $Recompose\ Tool$ was marked as a TOOL). If a menu or tool match was identified, a second pass through the text marked all parameters related to those items as PARAMETER matches (in those tutorial steps were the original match was found). This two-step process ensured that common words were not likely to be marked as parameters when in the wrong context. For example, *shape* is a common word that appears throughout tutorials but is often a parameter in the context of the *Lens Blur* command.

Though we leave this to future work, we note that there is an opportunity to improve the extraction through joint-inference [22] (e.g., the strings *10px*, *radius*, and "$Filter > Blur > Box\ Blur$" appearing near each other can be mutually reinforcing).