# Interview with Greg Flynn about KlaviyoCLI

Author: Eric Silberstein

Claps: 155

Date: Nov 29, 2022

Say you joined the *Reporting* engineering team at Klaviyo two weeks ago. You donâ€™t know all that much yet! Your team is in the middle of an incident. Your boss thinks something might be going on with the events system and asks you, the one person not looking through the logs or code, to page the *Event Query* team.

But how do you page them? Well, in your two weeks so far, KlaviyoCLI has been pretty useful. Try that!

```
>  klaviyocli --help
Usage: klaviyocli [OPTIONS] COMMAND [ARGS]...

  KlaviyoCLI

Options:
  --help  Show this message and exit.  [default: False]

Commands:
  aws         Commands for interacting with Klaviyo's Amazon Web Services.
  black       Run the python world's most opinionated linter.
  codeowners  Commands for verifying file ownership
  config      Configure tools used by klaviyo-cli
  database    A collection of tools to manage interacting with databases.
  deploy      Deployment commands for select applications
  docker      Wrapper for docker related commands
  git         Common commands for git repositories
  grafana     Wrapper for Grafana related commands
  help        Extended help and configuration
  ica         Client for platform-sre-team's Infrastructure Control API
  list        List Klaviyo objects
  pd          Commands for interacting with PagerDuty.
  scp         Perform a copy operation on a Klaviyo host using scp(1).
  ssh         Open an ssh(1) session to a Klaviyo host
  team        Team specific CLI modules
  terraform   Provides arguments needed to run terraform.
  tp          Commands for submitting tickets to TargetProcess
  tunnel      Open a tunnel to one of the pre-configured hosts
  update      Update KlaviyoCLI
  zk          Commands for interacting with Zookeeper
```

You go down the listâ€¦yes, pd, *Commands for interacting with PagerDuty.*

```
> klaviyocli pd --help
Usage: klaviyocli pd [OPTIONS] COMMAND [ARGS]...
```

```
  Commands for interacting with PagerDuty.

Options:
  --help  Show this message and exit.  [default: False]

Commands:
  active                      Activate a service, taking it out of...
  create-email-integration    Add a generic_email_inbound integration to an.
  create-rotation             Creates PagerDuty rotation and all required...
  incidents                   List all the current incidents
  maintenance                 Create a maintenance window on the given...
  page                        Page the person on call for the chosen service
  schedule                    Get your current and upcoming pagerduty...
  services                    Get all the services and who's currently on...
```

Okay, *page* looks rightâ€¦

```
> klaviyocli pd page
Usage: klaviyocli pd page [OPTIONS] SERVICE SUMMARY...
Try 'klaviyocli pd page --help' for help.

Error: Missing argument 'SERVICE'.
```

But whatâ€™s the name of the service? Ooh, letâ€™s use *servicesâ€¦*

```
> klaviyocli pd services
Service Id    Name                  Alias                  Status    On Call
------------  ------------------    -----------------      --------  -----------
...
PHXXXXX       Event Analytics       event-analytics        critical  Jxxx Pxxx
P5XXXXX       Event Pipeline        event-pipeline         active    Axxx Lxxx
P8XXXXX       Event Query           event-query            active    Dxxx Kxxx
PEXXXXX       Flows                 flows                  active    Dxxx Sxxx
...
```

Now you know what to do!

```
> klaviyocli pd page event-query Researching critical incident. Need help
```

Now if youâ€™re not an engineer youâ€™re thinkingâ€¦thatâ€™s insane. Isnâ€™t there some web page I can go to do this? And of course there is. But if youâ€™re an engineer, you might be like, this is incredibleâ€¦so many of the things I need are available in a command line tool with clear documentation and error messages.

And thatâ€™s exactly what KlaviyoCLI is. It replaced and/or wrapped a whole mess of other tools and documentation and made developer life at Klaviyo way more pleasant. A few things I like about it:

- **Install process** â€" checks for prerequisites and configures your environment
- **Updates itself** â€" zero friction to get the latest version or fear that youâ€™re out of date when running critical commands
- **Documentation** â€" clear and consistent
- **Error messages** â€" helpful and often tell you exactly how to fix the problem
- **Environment variables** â€" clarity on what environment variables need to be set for various commands

• **Extensible** â€" easy to add team-specific commands

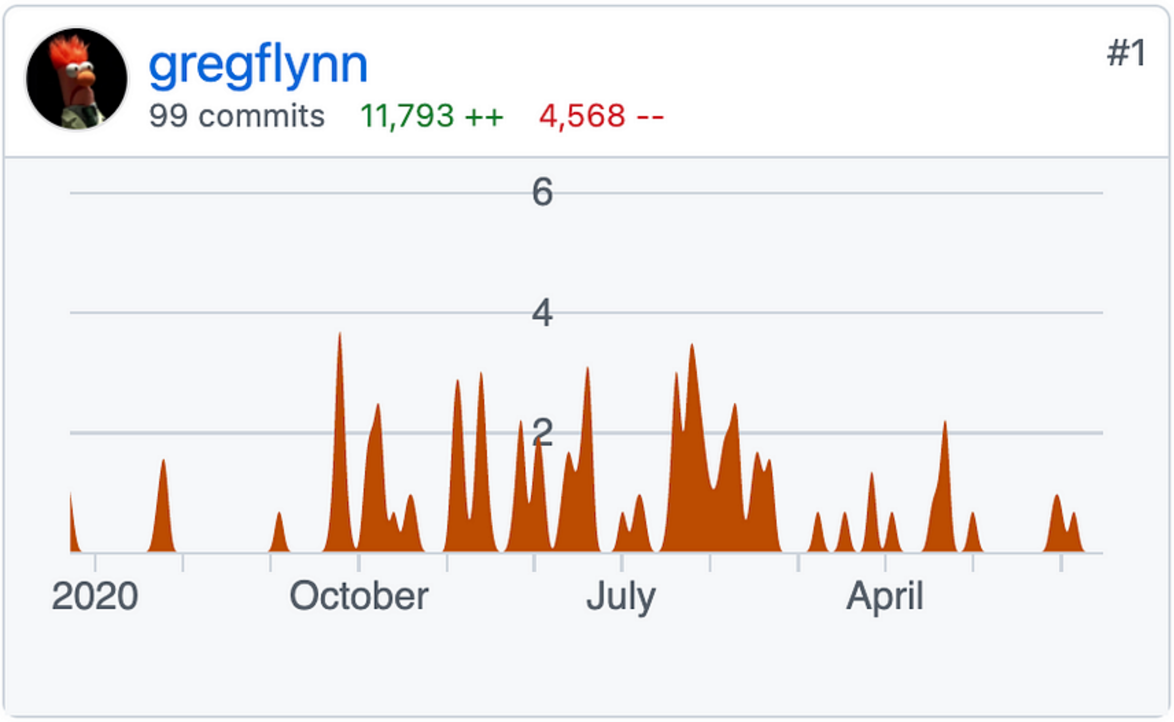
Our KlaviyoCLI repo has 107 contributors. The first and top is Greg Flynn from our SRE Velocity team:

```
> git log --reverse
commit 314fda5f0a32ac36ff8070e18760358300441abb
Author: Greg Flynn <gregflynn@users.noreply.github.com>
Date:   Tue Dec 10 13:59:08 2019 -0500

    Initial commit

commit 09ed699f20ab82ef5dc199d79745067f2025f8c5
Author: Greg Flynn <gregflynn@users.noreply.github.com>
Date:   Fri Mar 13 13:32:44 2020 -0400

    proof of concept using invoke as a cli library (#1)
```



In December, after Black Friday / Cyber Monday, and after five years at Klaviyo, Greg is leaving for his next adventure. When I found out, I wanted to ask him about the technology and design principles behind KlaviyoCLI, especially since the foundation has worked out so well.

(Interview edited for clarity.)

**How did this project start? Was the goal to create a single command line tool to do everything?**

No. That vision came later. We had a bunch of tools for deployment and infrastructure management built on fabric. When we moved Klaviyo from Python 2 to 3 we needed to migrate those tools to something new, the default being fabric plus invoke since fabric itself split off into those two projects. So our initial goal was limited to consolidating and improving those original set of tools, and doing so in a way that would give us a deliberate way to migrate to Python 3.

**What command line tools out there do you think are really good? Did you take inspiration from them?**

This isnâ€™t going to come as a shocker to you. Mostly the GNU tools. Iâ€™ve been a linux script kiddie since eighth grade. Inspiration from a couple angles. One is the GNU and open source tooling â€” the way command line arguments are built in that ecosystem. If somethingâ€™s a required argument, take it as an argument as opposed to a flag or option. With our old fabric tools pretty much everything was a flag or option. But with GNU you have more arguments. Other aspect is man pages. Itâ€™s so easy to look up extensive documentation for GNU tools. Having to leave your terminal to look up documentation is flawed UX.

**Are there design principles we stuck with?**

The biggest internal practical change from how we used to do things to KlaviyoCLI is that stack traces arenâ€™t an acceptable way to alert users of an error. With our old tools, if you didnâ€™t set up the right arguments, you would get a stack trace. But in KlaviyoCLI, we focus more on the user experience â€” the end user in the terminal. We give helpful error messages.

The other big design philosophy that I pushed is, taking the idea that stack traces arenâ€™t good error messages one step further, missing configuration should not just tell you the configuration is missing. It should tell you how to configure it.

For example, we have these @env_var_option decorators and these are generic wrappers to shared piece of configuration or secrets across commands, and there are unit tasks in KlaviyoCLI CI that make sure all the instructions to tell a developer how to populate that configuration are present.

For instance, the *prs* command requires GITHUB_TOKEN and GITHUB_USERNAME:

```
@env_var_option(Env.GITHUB_TOKEN)
@env_var_option(Env.GITHUB_USERNAME)
def prs(github_token, github_username):
    """Show your open pull requests on GitHub"""
    from github import Github
...
```

If you were the first person to write a command that needed that, CI would also make sure you added the instructions for developers:

```
_instruction_builder(
    Env.GITHUB_TOKEN,
    "token",
    "Unable to find GitHub personal access token",
    (
        "You can create a personal access token on GitHub at "
        "https://github.com/settings/tokens and selecting Generate New Tok
        "permissions for 'Repo' and 'Workflow', Generate Token, and then e
    ),
)
```

Hereâ€™s what it looks like if a developer runs the *prs* command without the GITHUB_TOKEN environment variable set:

```
> klaviyocli git prs
Unable to find GitHub personal access token
```

```
The token can either be specified by environment variable (GITHUB_TOKEN)
or by specifying it with --github-token. It is strongly recommended to set
value to an environment variable in your bashrc/zshrc/profile/etc.


You can create a personal access token on GitHub at https://github.com/set
```

**So someone working on KlaviyoCLI canâ€™t register a new environment variable without also writing the documentation for how to configure it?**

Correct. And then as an end user, if I want to see where things are used, we can actually introspect across all our commands and this spits out where everything is used. This is a huge part of making the CLI UX really good.

```
> klaviyocli help

KlaviyoCLI Environment Variables

Run with --show-instructions for additional info on missing variables

Variable                  Set    Usages
------------------------  -----  ------------------------------------------
GITHUB_TOKEN              âœ…     git, git.audit, team.velocity.mapping
GITHUB_USERNAME           âœ…     git
KL_AWS_KEY_NAME           â�Œ     aws.emr
...
PAGERDUTY_API_TOKEN       âœ…     pd, team.integrations.sentry_tp_automati
...
```

Requires_vpn is another good example of being able to give an actionable resolution. Itâ€™s so much better for the end user to be told their VPN isnâ€™t on rather than getting a generic connection error. These decorators allow for good reusable UX patterns.

```
@requires_vpn
...
def watch(...):
    """Watch the progress of an ongoing deploy"""

def requires_vpn(f):
  @functools.wraps(f)
  def wrapper(*args, **kwargs):
    if check_vpn():
      return f(*args, **kwargs)
    else:
      error(
        "You must be connected to the Klaviyo VPN for this command to exec
      )

  return wrapper
```

**For both these principles â€” showing proper error messages instead of stack traces and providing configuration instructions â€” itâ€™s a lot more work. How big should your engineering team be before itâ€™s worth doing these things?**

Itâ€™s less about headcount and more about who the end user is. When we had our old fabric tools, all our end users were pretty much SREs except for the app deployment stuff. But KlaviyoCLI everyone uses. So I think the threshold is â€” is it an internal tool or an external tool. And when it becomes an external tool beyond your team, youâ€™re going to have users who then donâ€™t know what that stack trace means.

**It seems very fastâ€¦**

Delayed imports. Delaying heavy imports to within command scopes. It speeds up initial load time of all your commands, but the real unlock is it speeds up the ability for your entire command framework to be parsed. If I want to go generate bash or zsh completions, I just need to know the tree of all available commands.

**Letâ€™s talk about how it automatically updates itself. Whatâ€™s the idea? Why is that important? How did we build it?**

I took a lot of inspiration from how progressive web apps work around swapping the code out underneath and rerunning. It was one of the early but critical features of the framework, because if folks arenâ€™t able to update and arenâ€™t able to get new features it makes it harder for us to roll things out. We can also do a check to see if your KlaviyoCLI is out of date before you run a command.

**Do we only do that for some commands?**

Correct. Itâ€™s an opt-in basis. And you can also opt-in conditionally meaning you can give the user the option to bypass. Take a look at the decorator. Here the user will be required to be current:

```
@ensure_klaviyocli_up_to_date()
â€¦
def deploy(
â€¦
```

Here they will get a warning:

```
@ensure_klaviyocli_up_to_date(allow_bypass=True, allow_dirty=True)
â€¦
def zkda(
â€¦
```

If someone is kicking off an app deploy, and say there was a bug fix in our deployment command, we want to make sure they have it.

**Right. This is huge. I remember a few years ago, things started changing so quickly, and it was nerve-racking never being sure you were following all the latest instructions and had your environment exactly right for something as critical as a deploy or running terraform.**

Exactly.

**After it updates itself it then runs the command again right?**

Yes. Through sys.argv we get the name of the command and all the arguments. So thatâ€™s how we know what they were trying to do. And the ensure_kcli_up_to_date wrapper is going to run before all of its wrapped functions because we call f() so far down.

```
def ensure_klaviyocli_up_to_date(allow_bypass=False, allow_dirty=False):
  """Wrapper for click commands to check to make sure that klaviyocli is u
  before executing the desired command.

  NOTE: there is nothing stopping a user from commenting out this decorato
      local and running the command anyways, this is not a substitute from
      roles/rights/validation.

  Args:
      allow_bypass: Set to True to display a warning and prompt, allowing
          still run the command instead of exiting
      allow_dirty: Set to True to allow running the command when the Klavi
          checkout is dirty instead of exiting
  """

  def ensure_kcli_up_to_date(f):
    @functools.wraps(f)
    def wrapper(*args, **kwargs):

    ...

      if not repo.is_up_to_date():
        warn("KlaviyoCLI is not up to date")
          if click.confirm("Run KlaviyoCLI update?"):
            repo.update()
            requires_reinvoke = True
          elif allow_bypass:
            if not click.confirm("Continue?"):
              error("Aborted")
          else:
            error(
              "This command requires KlaviyoCLI update; please run `klaviy
            )
    ...

        if requires_reinvoke:
           click.secho("Re-invoking klaviyocli with updates", fg="green")
           os.execlp("klaviyocli", *sys.argv)
           # we bomb out of this code execution here because of os.exec re
           # the current process with the new one we specified

        f(*args, **kwargs)

     return wrapper

  return ensure_kcli_up_to_date
```

**What if, say, it needs to install new dependencies?**

Thatâ€™s all handled in repo.update() because that runs install.sh. It will even upgrade the python version if needed, and still then execute the original command.

```
class KlaviyoCLIRepository:
  ...
  def update(self):
    click.secho("Updating...", fg="yellow")
    self._handle_timeout(self._repo.remote().pull)
    subprocess.check_call([Path(get_klaviyocli_home()) / "install.sh"])
```

**Can you talk about install being idempotent? What was the reason and how hard was it to do?**

Two primary reasons to make install.sh idempotent. One is dogfooding. For me, itâ€™s easier for my new installs and existing installs to go through the same code and checks. Other is user experience. Itâ€™s harder to make the code idempotent. No one would disagree with that. All that extra effort is for user experience. Goes back to that same design philosophy. None of the python code should produce a stack trace on error. So â€” much harder to do error checking and best practices in bash and so we want to keep install.sh as small as possible â€" but same design philosophy. If we detect for instance that you donâ€™t have 1Password CLI set up, boom, exit 1, go to this URL and set it up:

```
if ! command -v op > /dev/null; then
   echo "1Password CLI is not installed, please install it from:"
   echo "\t https://1password.com/downloads/command-line/"
   exit 1
fi
```

**How do we distribute KlaviyoCLI?**

In the first, first version you would pip install it as a package, which worked for a little while, but there were issues. Now you install by doing a git clone and then our install.sh script handles everything else going forward. A big unlock from managing things ourselves thatâ€™s harder in the pip installable world is we can take all this non-python code down with us and manage it in a dedicated directory. We also manage binaries like saml2aws and multiple versions of terraform which get pulled down from an S3 bucket.

**Why Click instead of Invoke? Looking at the git history I see you started with Invoke. Why the transition to Click?**

It took some convincing for me but in the end Iâ€™m really glad I was convinced to switch it all over to Click. The biggest thing with fabric and invoke is theyâ€™re a bit more of magical frameworks. Itâ€™s a lot harder to follow where the code execution is getting handed off to various pieces. They make use of globals quite often, so you import some global, you access it in your command which changes state. I find that to be a harder paradigm to follow.

What Click offers us is 1) itâ€™s instantly familiar if youâ€™ve ever used a microframework in Python like Flask or Sanic because itâ€™s heavily decorator forward, and 2) because of that if you are a Python developer who is familiar with the decorator pattern you can also follow. A lot of really good accessibility wins from that perspective. And because itâ€™s less dependent on global state, itâ€™s easier to write utilities for.

The other upside to Click is superior documentation and superior command line tools. Click is the only Python library that any time anyone comes to me and is like â€" *I want to learn how to write KlaviyoCLI commands*, I point them to Clickâ€™s documentation and say expressly I have read every single page of this documentation and every single page is absolutely amazing. There is useful information on every line of their documentation. From all the various ways of doing options, options groups, arguments, progress bars. Itâ€™s a really well-documented, well-scoped library.

**How did we get adoption?**

Just a question of putting juicy enough carrots into the tool. We started off with smaller tools to scope the feedback and iterate. If thereâ€™s something thatâ€™s obviously a core user experience problem you donâ€™t need a hundred people telling you that. So iterating on it with small commands early. And then the big command was app deployment. Once app deployment got moved over and it was faster and more reliable to deploy through KlaviyoCLI, that was the catalyst and 90+% of engineering installed it. That was then the stepping stone to leverage it more and more ways. Now we recommend KlaviyoCLI in the developer environment before you install app or other repos because it automatically installs the pip conf:

```
if [[ $ONE_PASS_CLI_VERSION == 1* ]]; then
  ONE_PASS_DOC=$(op get document 'KlaviyoCLI pip.conf')
else
  ONE_PASS_DOC=$(op document get 'KlaviyoCLI pip.conf')
fi
if [[ "$ONE_PASS_DOC" == "" ]]; then
  echo "Error getting pip.conf from 1Password."
  exit 1
else
  mkdir -p $(dirname "$PIP_FILE")
  echo "$ONE_PASS_DOC" > "$PIP_FILE"
fi
```

**How does the teams stuff work? What are teams allowed to do if they want to introduce commands that are only for them?**

We identified that use case fairly early because Velocity had that use case. We have commands that we wanted to generate that had a lot of Velocity jargon in them, or we didnâ€™t want people to think they should run willy-nilly, so we kind of hide them. We give folks a place to put stuff that isnâ€™t necessarily applicable outside of their team. Thereâ€™s a team module:

```
> klaviyocli team
Usage: klaviyocli team [OPTIONS] COMMAND [ARGS]...

  Team specific CLI modules

Options:
  --help  Show this message and exit.  [default: False]

Commands:
  datascience   Datascience team CLI commands
  integrations  Integrations team CLI commands
  sre-infra     SRE Infrastructure team CLI commands
  velocity      Velocity team CLI commands
```

You can see for Velocity we got a bunch of commands for things like managing Jenkins agents. Nobody outside of Velocity should do that.

```
> klaviyocli team velocity
Usage: klaviyocli team velocity [OPTIONS] COMMAND [ARGS]...

  Velocity team CLI commands

Options:
  --help  Show this message and exit.  [default: False]

Commands:
  artifactory-usage        Audit artifactory package usages
  github-user-to-slack-id  Map GitHub login to slack ID with their name...
  jenkins-agent            Jenkins-Agent maintenance commands
  reassign                 Reassign a bug ticket to another team
  sot                      Commands for Scale-Out-Test related items
```

**Any final thoughts?**

Itâ€™s exciting to see the amount of contributions you get when you make the barrier to entry low and the ability to follow standards and best practices really easy.

## â€œCustomerâ€� quotes

I posted the following in our #dev-team channel.

*Please thread your thoughts on klaviyocli. Weâ€™re doing a blog post interview with @greg.flynn about the design principles behind klaviyocli and would be nice to include customer quotes. Customer = the people in this channel!*

- â€œI donâ€™t often think about it because it just works for most of the things I need it for most of the time. Many of the utilities people have added to it definitely turn tasks that used to be really manual/painful/time-consuming into fast and easy commands.â€� â€” Woody Austin
- â€œIt makes on-boarding to Klaviyo and getting new systems set up for the first time much less intimidating. Instead of spending a lot of time figuring out exact syntax for how to get something to run, you can focus just on inputs and the big picture of what is happening. It also feels nice that teams at Klaviyo put lots of careful thought into these things to make the rest of our lives easier!â€�â€” Lilia Staszel
- â€œExtending the tool for team-specific needs is easy and convenient. There are many utilities to help the tool developers (e.g. it has a standard way to pass configuration parameters to commands as either environment variables or explicit flags). Having a centralized tool means that team members donâ€™t have to install a new cli for each use case and reduces friction with other teams for the same reason.â€� â€” Vinicius Aurichio
- â€œEchoing Viniciusâ€™s point, tools like this are often grab-bag composites of different requirements, needs and user experiences, amended whenever someone needs to hang new functionality off the shared framework (what a mentor of mine dubbed â€˜Christmas tree codeâ€™). Instead of being fractured and hard to contribute to, it is open and extensible. Thatâ€™s a testament to the good architectural foundations laid early on and the quality of contributors and contributions.â€� â€” Jason Panzer
- â€œHaving a centralized tool is really nice. There is less to remember and by having it in one place with all commands documented makes things easy to discover and learn how to use.â€� â€” Tyler Bream
- â€œTransitioning from having one-off commands that one needed to search in their history or hunt for in documentation to having a tool that makes it easier to discover and

understand usage of these commands (using standardized help options) has been great for engineering productivity.â€� â€” Vedant Puri

- â€œItâ€™s fun to try out the different commands and see what colorful text and emoji gets printed out on the consoleâ€¦and +1 to it being useful.â€œâ€” Suf Hamzah