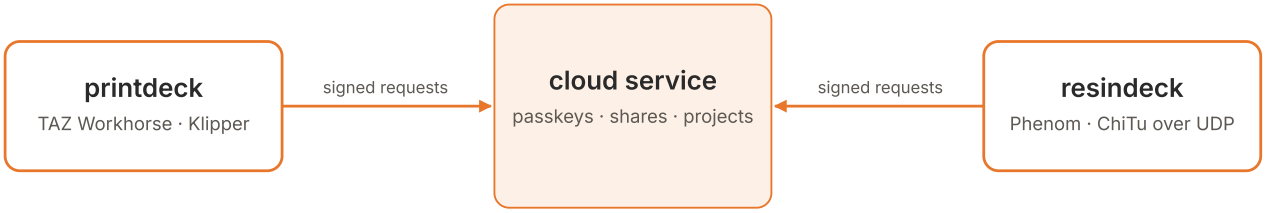
PRINTDECK · TECHNICAL NOTES

Printdeck

Architecture, identity and passkey sign-in of the decks and the cloud service at printdeck.beetlebug.org

This document describes the parts of the Printdeck system and how they connect. It explains how each deck, person, page and GitHub Action proves who it is. It shows why the passkey sign-in and the builds from GitHub are secure, and where their limits are.



1 November 2026 · code at commit c6215969

Jeremy Collins

Contents

1	Overview	the decks, the printers, the slicers, the cloud and `pwf`
2	Identity and trust	decks, the owner, members, share links, upload grants
3	Passkey flows	owner enrollment, the invite workflow, sign-in, the session token
4	Why it is secure	each control, where the code has it, and the limits
5	The data of a print	FDM and resin: live tile, share page, project record, notes, photos, the project page, the 3D view
6	Builds from GitHub	plans, plates, the slice runs, the approval, the profiles, forks, security
7	The print queue	the queue of a deck, files from slicers, plates of builds, what a project sees
8	The apps	the deck API for apps, the iPhone and Watch app, the sign-in in the app
9	Updates	the A/B image, staged updates, restarts during a print
A	API routes	each route of `/api/v1/` and its auth

Sources. The facts come from the code at commit c6215969: `cloud/api`, `cloud/deploy/template.yaml`, `cloud/cmd/pwf`, `cloud/cmd/pwf-web`, `cloud/deploy/web/assets`, `deck/internal/cloud`, `deck/internal/chitu`, `deck/internal/printer`, `deck/cmd/printdeck`, `deck/internal/system`, `os/` and `profiles/`. The builds from GitHub also come from `docs/specs/github-builds.md` and from the repository `beetlebugorg/printdeck-action` at commit 4f1e056. The passkey library is `github.com/go-webauthn/webauthn` v0.18.2.

Words. A *deck* is a Raspberry Pi 5 with a touch screen that runs Printdeck OS. The *owner* is the person who runs the service. A *member* is a friend that the owner invited to one or more projects. The *API* is the Go program on AWS Lambda. The *bucket* is the S3 bucket of the service. The *Action* is `beetlebugorg/printdeck-action` in a workflow of a project repository on GitHub.

1 Overview

Two decks drive two printers on the home network. One cloud service shows their prints to the owner and to friends.

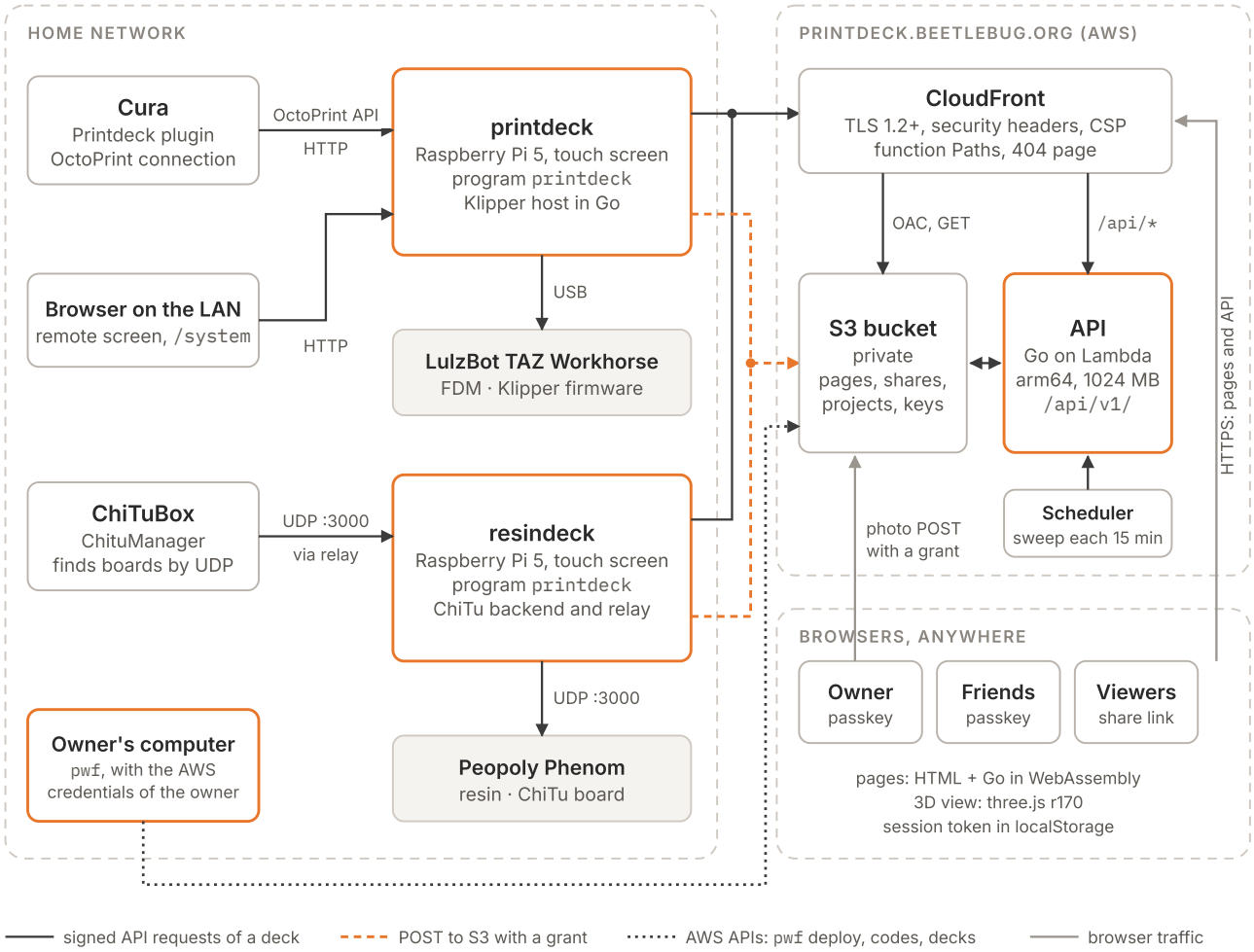


Figure 1. The parts of Printdeck. The decks connect out to the cloud. The cloud never connects to a deck. `pwf` runs on the computer of the owner and uses the AWS APIs directly.

1.1 The decks

Each deck is a Raspberry Pi 5 with a touch screen. It runs Printdeck OS, a small read-only Linux system that Buildroot makes. The OS has no package manager, no shell and no SSH. The program `printdeck` is one static Go binary. It draws the touch screen, serves a remote screen at `http://<deck>.local`, and keeps the print history.

- **printdeck** drives the FDM printer, a LulzBot TAZ Workhorse. `printdeck klippy` is a Go port of the Klipper host. It talks to the Klipper firmware on the printer board over USB. The deck talks to the Klipper host on its API socket. Slicers send files through the part of the OctoPrint API that Cura, OrcaSlicer and PrusaSlicer use.
- **resindeck** drives the resin printer, a Peopoly Phenom. The ChiTu backend sends one line of text in one UDP packet to port 3000 of the ChiTu board. The board sends one line back. The deck also runs a relay on its own UDP port 3000, so ChiTuBox (its ChituManager) lists the deck as a printer.

The relay answers the discovery broadcast of ChituManager (`M99999`) with the line of the board. It keeps each upload (`M28`, the data packets, `M29`) as a file on the deck. ChituManager sends its data packets with no wait and does not resend them. So the relay answers each packet itself, and then the deck sends the file to the board one packet at a time. The deck keeps a copy of each sliced file. That copy gives the layer plan, the layer pictures and the 3D cells of a resin print (section 5).

1.2 The cloud service

The service is at `https://printdeck.beetlebug.org`. It is serverless, in one AWS CloudFormation stack in us-east-1. A second, older stack, printdeck-relay, keeps the secret of the push relay and a relay function for old copies of the iPhone app. The owner makes the stack with `pwf deploy` on their own computer (section 1.3).

- **CloudFront** is the only public entry. It sends `/api/*` to the Lambda function URL with no cache. It serves all other addresses from the bucket. It adds the security headers and the Content-Security-Policy to each reply.
- **S3** keeps all the data as files in one private bucket. CloudFront reads only the listed keys, through Origin Access Control (OAC): the pages, the shares, the public pages, the pictures of the comments on pull requests (`gh/*`), and the files of the projects behind a signed cookie. The API reads and writes all keys.
- **The API** is one Go program in one Lambda function, `printdeck-cloud-api` (arm64, 1024 MB, 60 s). It serves `/api/v1/`, and also the push relay of the iPhone app at `/api/v1/relay/`. A schedule runs its sweep every 15 minutes to delete or move old shares. For the jobs after a request (the comments on pull requests and the checks of their state), the function invokes itself with an event. A failed job does not run again. The API reads the records of a project 64 at a time, keeps the name of the owner in memory for 5 minutes, and writes one log line for each request with the route, the time and the count of bucket calls.
- **GitHub Actions** of a linked repository sends the plans and the builds of the project to the API, with an OIDC token of GitHub (section 6). With the GitHub App, the API also starts the slice runs and writes a comment on each pull request.
- **The pages** are static HTML. Their logic is Go, compiled to WebAssembly (`cmd/pwf-web`). `pwf` uses TinyGo when it finds it, because TinyGo makes a file about 5 times smaller. The 3D views use three.js r170 from the bucket.

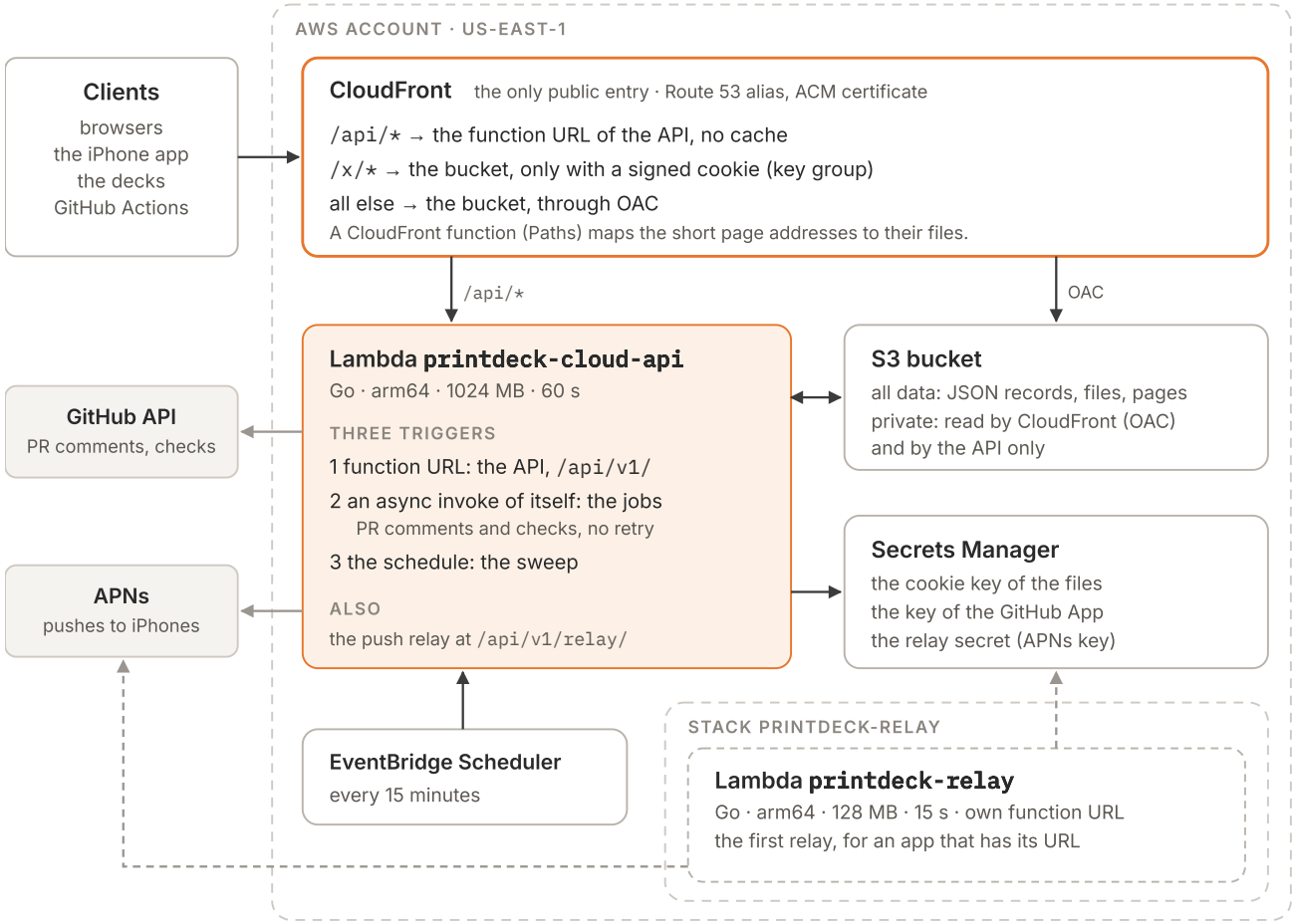


Figure 2. The AWS pieces of the service. Two Lambda functions run Go on arm64. `printdeck-cloud-api` does all the work of the service: the API, the jobs after a request, the sweep, and the push relay. At a cold start it reads its 3 secrets at the same time. `printdeck-relay` is the first relay, in its own stack. It shares the relay secret, so the send tokens of the apps stay valid.

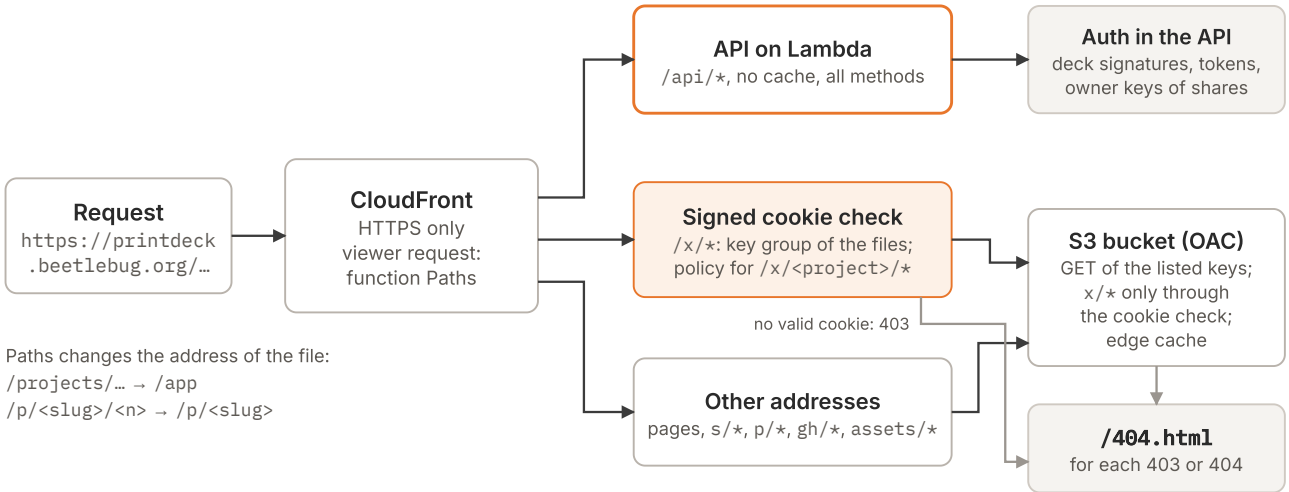


Figure 3. How CloudFront routes a request. The API never replies 403 or 404. A 403 or 404 from S3 or from the cookie check shows the "not found" page with status 404.

1.3 The owner tool `pwf`

`pwf` (`cloud/cmd/pwf`) is a command-line tool. The owner runs it on their own computer, with the AWS credentials of the owner. It does not run on a deck, and it does not run in AWS. It uses the AWS APIs directly: CloudFormation, S3, Lambda, Secrets Manager, Route 53 and STS. It does not use CloudFront or the API of the service.

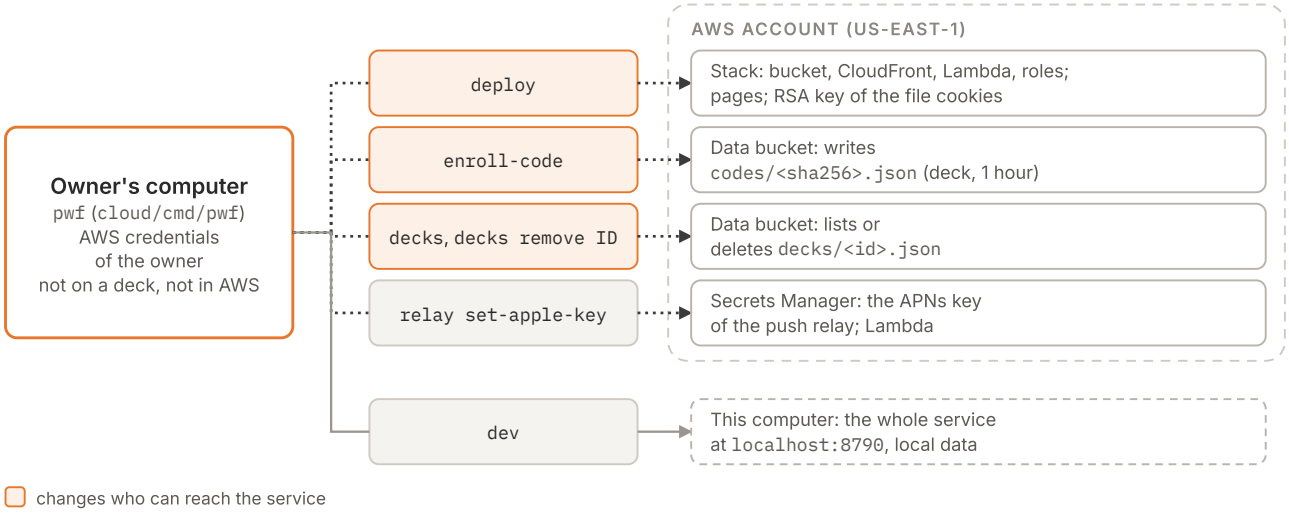


Figure 4. Where `pwf` runs, and what each command reaches. The orange commands change who can reach the service.

Command	What it does	Effect on the auth
`pwf deploy`	Builds the Lambda function (Go, linux/arm64) and the WebAssembly pages. At the first deploy, makes an RSA key pair for the file cookies and keeps the private key in the secret `<stack>-files-key`. Makes or changes the CloudFormation stack with the public key. Uploads the pages to the data bucket, and writes the pages of the running shares again. For each public project, it runs the whole publish again, so it also repairs a publish that stopped before its end.	Makes the private data bucket, its bucket policy, CloudFront with the CSP and the key group of the files, and the role of the function. The API makes `keys/session` in this bucket at its first use.
`pwf enroll-code`	Writes `codes/<sha256>.json` straight to the data bucket, and shows the code. The code works once, for 1 hour.	The only way to get a deck enrollment code. It uses the AWS rights of the owner, so the code does not go through the API.
`pwf decks`, `pwf decks remove ID`	Lists the files in `decks/`, or deletes one.	A removed deck gets 401 at its next request.
`pwf relay set-apple-key`	Writes the APNs key of the push relay into Secrets Manager, and makes the functions start again to read it.	No effect on the sign-in.
`pwf github-app --id N --key app.pem`	Writes the App ID and the private key of the GitHub App into the secret `<stack>-github-app`. The next `pwf deploy` gives the secret to the function.	The function then starts the slice runs of the builds (section 6.6), and writes the comments on the pull requests. Each App token works for one repository, with only the permission of its call: Actions to start a run, Pull requests to read a pull request or to write a comment.
`pwf dev`	Runs the whole service on the computer at `localhost:8790`, with the data in memory or in a folder (`-data`). It shows a deck enrollment code and an owner link.	Needs no AWS credentials. It has its own data and its own `keys/session`.

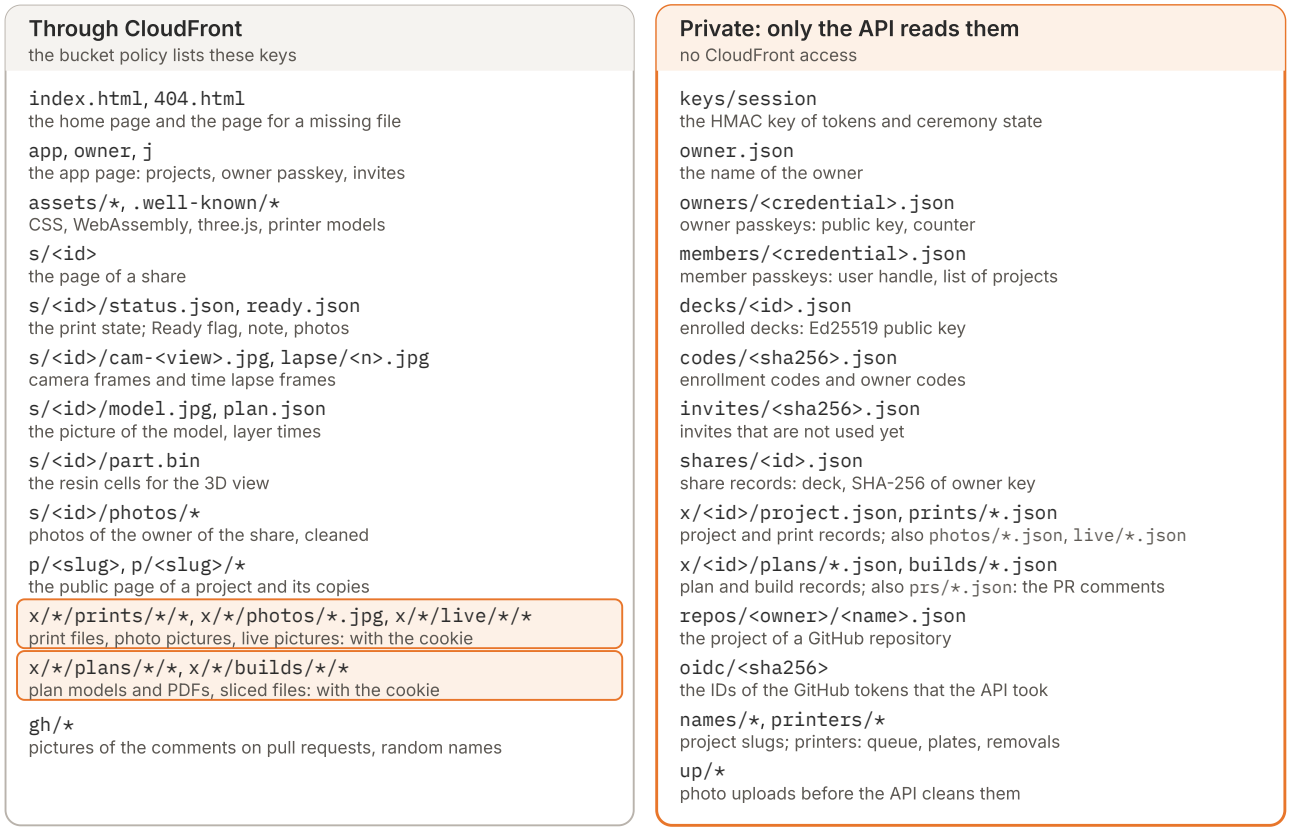


Figure 5. The layout of the bucket. The bucket blocks all public access. CloudFront can read only the keys on the left. The keys on the right have the codes, the passkeys, the session key, the repository links, the used GitHub tokens and the state of the pull request comments.

2 Identity and trust

Each kind of actor proves who it is in its own way. No actor has a password. No deck has an AWS key.

2.1 The actors

Actor	Proof	The service keeps	Scope
Operator	AWS credentials, on the computer of the owner	(AWS IAM)	All of the stack and the bucket. Runs `pwf`: deploys, makes enrollment codes, removes decks (section 1.3).
Deck	Ed25519 signature on each request	`decks/<id>.json`: the public key	The rights of the owner in the project API. Also the actions that only a deck does: shares, prints, live tiles, owner codes.
Owner	Passkey, then a session token	`owners/<credential>.json`: public key, counter	All projects: make, rename, delete, invite, remove members, public address.
Member	Passkey, then a session token	`members/<credential>.json`: public key, counter, list of projects	The projects of their passkey: see each one, add photos, move or remove their own photos.
Share owner	The owner key of one share	The SHA-256 of the key in `shares/<id>.json`	One share: the Ready flag with a note, and up to 20 photos.
Action	OIDC token of GitHub, a new one for each request	`repos/<owner>/<name>.json`: the project of the repository; `oidc/<sha256>`: the used tokens	The plans and the builds of the one project that links its repository. The token of a member's fork sends the plan of the member's open pull request (section 6.8). Only the workflow `printdeck.yml` on main or master fills a build (section 6.7).
Viewer	Knows the share link	no record	Reads the share page and its files until the share ends.
Visitor	(none)	no record	Reads the public page `/p/<slug>` of a project, when the owner turns it on.

2.2 Deck enrollment

A deck gets its identity once. The owner runs `pwf enroll-code` on their own computer. The tool uses the AWS credentials of the owner and writes the SHA-256 of a random code to `codes/`, with the kind "deck" and an expiry of 1 hour. The owner types the code on the deck in Settings, Cloud. The deck makes an Ed25519 key pair and sends the code, its public key and its name. The API deletes the code file and then checks it. So a code works one time, also when the check fails.

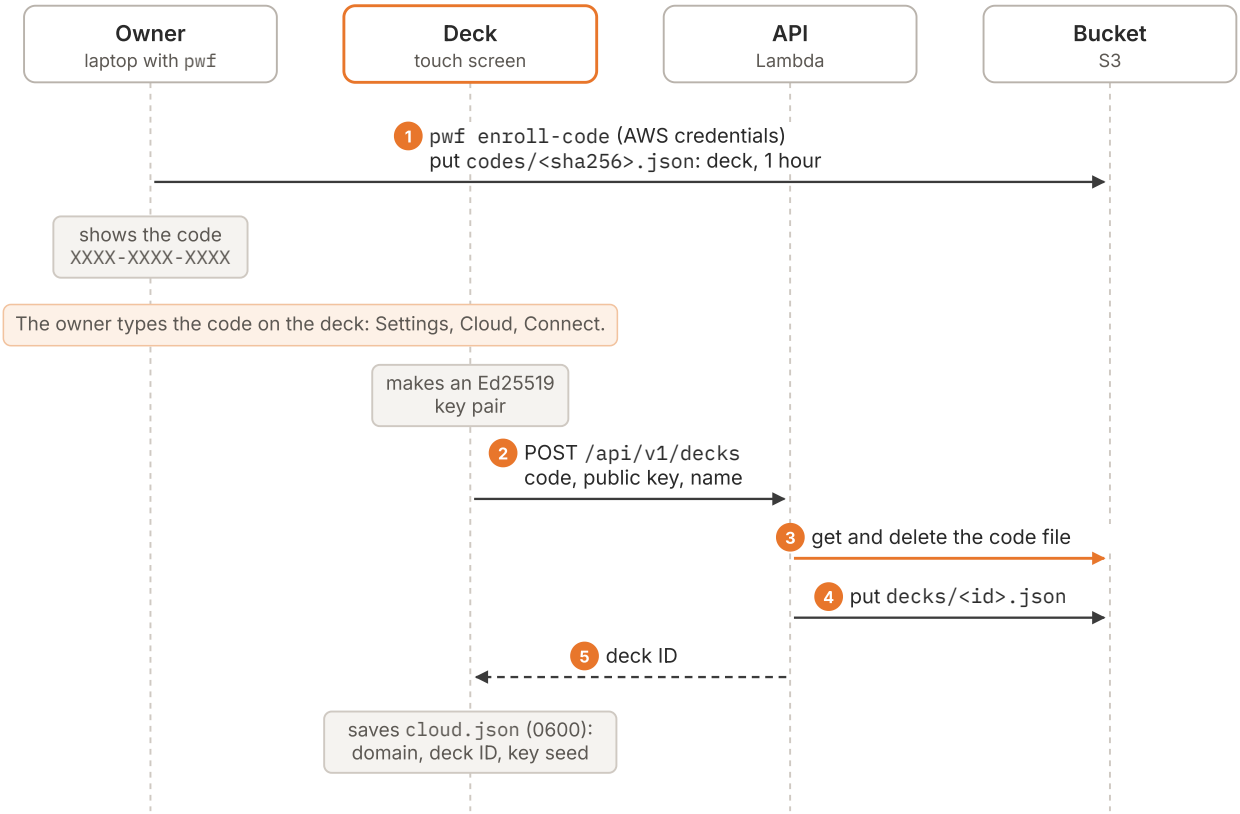


Figure 6. Deck enrollment. Step 1 goes from the computer of the owner straight to S3, with the AWS credentials of the owner. The private key never leaves the deck. The bucket keeps only the hash of the code and the public key.

A code has 12 characters from an alphabet of 31 (no 0, O, 1, I or L). That is about 59 bits. The deck keeps its identity in `/var/lib/printdeck/cloud.json` with mode 0600: the domain, the deck ID, the name and the 32-byte seed of the key. To end the access of a deck, run `pwf decks remove ID`. Disconnect on the deck also deletes its file at the service and on the deck.

2.3 Signed deck requests

A deck signs each request to the API. The signed text has six lines: a version, the method, the path with its query, the time, a random request ID, and the SHA-256 of the body. The four headers `X-Printdeck-*` have the deck ID, the time, the request ID and the signature.

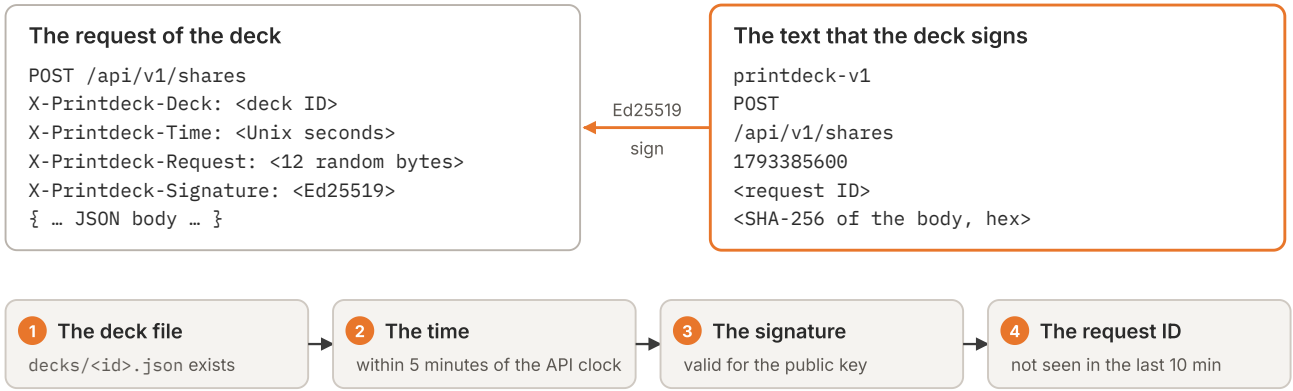


Figure 7. A signed request and the checks of the API (`cloud.Verify` and `Server.deck`).

The signature covers the body, so a changed body fails. The time check limits a replay to 10 minutes. Each instance of the function also keeps the request IDs that it saw in that time, and refuses a second use. A new instance starts with no IDs, so this check is partial. TLS is the main guard against a replay.

2.4 Projects and members

A project is `x/<id>/project.json`, with a random 13-byte ID. It has the members, the open invites, the prints and the share links. The owner makes a project in a browser or on a deck. The owner invites a friend with a first name (section 3.3). The friend joins with a new passkey, or with the passkey from an earlier invite. One passkey of a friend then works for all the projects that the friend joined. The owner can remove a member. The API then removes the project from each passkey of the member. It deletes a passkey file only when no project is left on it. The next request of the member on that project fails.

Members see the private files of the project through CloudFront, with signed cookies of that project (section 2.7). The bucket stays private, and no member gets a key to it.

2.5 Share links and the owner key

A share link shows one print while it runs. A deck makes it after a tap on Make the link. The API makes a share ID and an owner key, and keeps the SHA-256 of the owner key. The deck shows the link with the owner key in the fragment.



Figure 8. The owner link of a share. A browser does not send the fragment to a server. Copy link on the page gives the address with no owner key.

The page sends the owner key in the header `X-Share-Key`. The API compares the hash in constant time. The owner key works only for Ready and for photos. The share ends 24 hours after the end of the print, or 24 hours after Ready, whichever is later. The sweep then deletes the share. When the print is in a project, the sweep deletes the files of the share instead, clears the owner key, and makes the page send visitors to the print page. A share with no end goes after 3 days. A lifecycle rule of the bucket also deletes `s/` files after 3 days.

Share links are public by design

A share link works for anyone who has it, with no sign-in. Through CloudFront, a viewer reads `status.json`, `ready.json`, the camera and time lapse pictures, `model.jpg`, `plan.json` and `part.bin` of the share. The purpose is a print that friends with no account can watch. These things limit the exposure:

- The share ID has 13 random bytes: 21 characters in the link. CloudFront cannot list the bucket, so the link is the only way to the files.
- The files of a share have no passkeys, codes, owner keys or member names. They have the print state, the pictures, the printer name, the name of the job, and the Ready note and photos of the owner of the share.
- The sweep deletes the files 24 hours after the end of the print, or after Ready. A lifecycle rule of the bucket deletes `s/` files after 3 days.

- A share in a project names the project and links to it. The link goes to the public page when the project has one. Otherwise it goes to the project page, which needs a passkey of the project.

> **Possible future work (not built).** Two options came up:
>
> - Members-only shares: signed cookies on `/s/<id>/`, in the same way as the files of a project.
> - A choice for each share: public, or members of its project only.

2.6 Upload grants

Large files do not go through the API. The API makes a grant: an S3 POST policy that it signs with the credentials of the function. The deck or the page then sends the file straight to S3. S3 checks the policy and refuses a file that does not obey it.

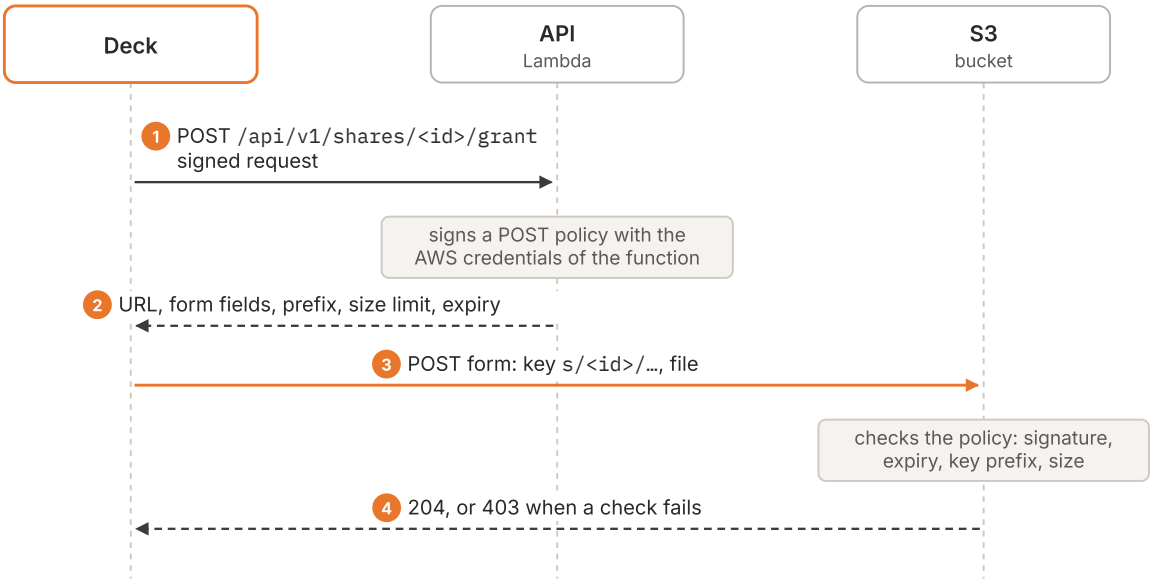


Figure 9. An upload with a grant. The deck has no AWS key. The policy limits the key prefix, the size and the time.

Grant for	Key prefix	Largest file	Works for	Asked by
Share files	`s/<share>/`	16 MB	1 hour	the deck of the share
Share photos	`up/<share>/<photo>/`	25 MB	15 min	the owner key of the share
Print files	`x/<project>/prints/<print>/`	8 MB	1 hour	a deck
Live picture	`x/<project>/live/<slot>-<pic>/`	8 MB	1 hour	a deck
Project photos	`up/x<project>/<photo>/`	25 MB	15 min	the owner or a member
Plan models and PDFs	`x/<project>/plans/<plan>/`	200 MB	1 hour	the plan run of the Action
Build files	`x/<project>/builds/<build>/`	200 MB	1 hour	the slice run of the Action

A photo goes to `up/` first. The API then decodes it and writes new JPEG files with no metadata: no EXIF data, no GPS position and no comments. A test checks this (`TestCleanPhotoStripsMetadata`). A lifecycle rule deletes files under `up/` after 1 day. The CORS rule of the bucket accepts GET and POST only from `https://printdeck.beetlebug.org`. Uploads use POST with a grant. Reads of project files use the signed cookies, on the same origin. GET serves the signed S3 links, which the API gives when the stack has no cookie key.

2.7 Files of the projects: signed cookies

The files of a project are under `x/<project>/` : pictures of models, layer plans, time lapse frames, photos and live pictures. CloudFront serves them at the same paths on the domain, under the behavior `/x/*` . That behavior accepts only requests with a valid CloudFront signed cookie. The bucket policy lets CloudFront read only five patterns under `x/` :

- `x/*/prints/*/*` : the files of a print, such as `model.jpg` , `plan.json` and `lapse/*.jpg` .
- `x/*/photos/*.jpg` : the pictures of the photos, also the small copies `-s.jpg` .
- `x/*/live/*/*` : the pictures of the live tiles.
- `x/*/plans/*/*` : the models of a plan, as STL files for the 3D views of the plan page, and the PDFs of the plan with a picture of each first page.
- `x/*/builds/*/*` : the sliced files of a build.

The JSON records stay with the API: `project.json` , `prints/<id>.json` , `photos/<id>.json` , `live/<slot>.json` , `plans/<id>.json` and `builds/<id>.json` . CloudFront cannot read them, also with a valid cookie.

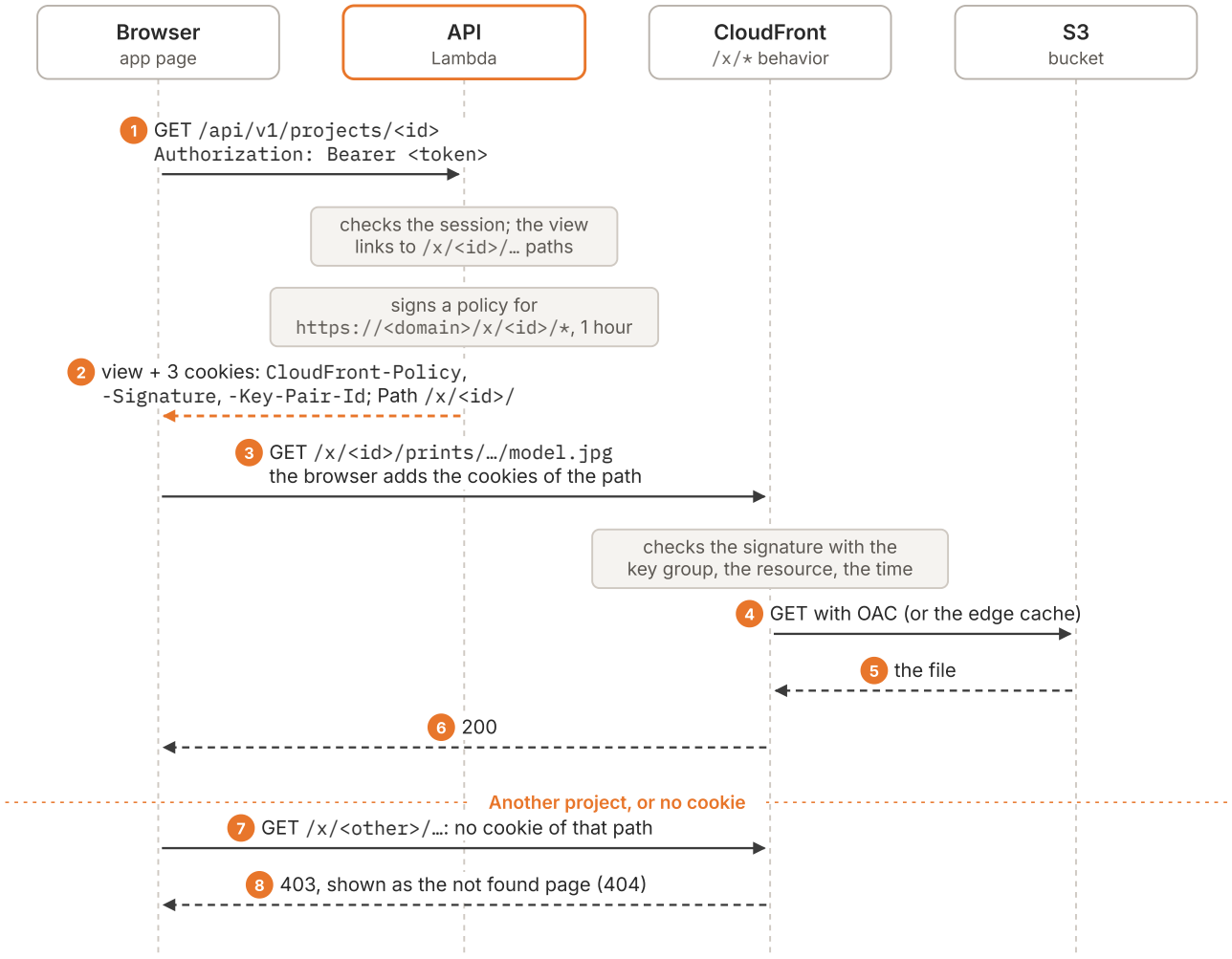


Figure 10. The signed cookies of the files of a project. The cookies of one project do not open the files of another project.

1. An API reply that links to files of a project gives the paths `/x/<project>/…` in place of signed S3 links. These replies are the project list (cover pictures), the project view with its live tiles and its PDFs, the view in the reply of `GET /api/v1/names/<slug>?view=1` , the time lapse frames of a print, a plan with its models and PDFs, and a build.

2. The API notes each project of the links. For each one, it signs a custom policy with the resource `https://<domain>/x/<project>/*` and an expiry of one hour. The signature is RSA with SHA-1, as CloudFront requires.
3. The reply sets three cookies for each project: `CloudFront-Policy`, `CloudFront-Signature` and `CloudFront-Key-Pair-Id`. Each has `Path=/x/<project>/`, `Secure`, `HttpOnly`, `SameSite=Lax` and an expiry of one hour. The Lambda function sends them in the cookie list of the function URL reply, because the headers keep only one `Set-Cookie`.
4. The browser sends the cookies only with requests under that path. CloudFront checks the signature with the public key in the key group `<stack>-files`, the resource and the time. Then it reads the file from S3 with OAC, or from its edge cache.

The owner tested the live service. A request with no cookie gets 404. With the cookie of the project, `plan.json`, `model.jpg` and photo pictures get 200, and `project.json`, `prints/<id>.json` and `photos/<id>.json` get 404. The same cookie on another project gets 404. The private key is in the secret `<stack>-files-key`. The function reads it at its start (`FILES_KEY_SECRET`, `FILES_KEY_ID`), at the same time as the secrets of the GitHub App and of the relay. When the stack has no key, the API gives signed S3 links for one hour, as before. A deck has no cookies, so the API always gives a deck a signed S3 link of one hour for each plate file that waits for it.

2.8 Who can do what

Action	Deck	Owner	Member	Share owner	Viewer
Make a share link, upload its files	✓	·	·	·	·
See a share page	✓	✓	✓	✓	✓
Mark a share Ready, add share photos	✓	·	·	✓	·
Ask for an owner code	✓	·	·	·	·
Make, rename or delete a project	✓	✓	·	·	·
Invite a friend, remove a member	✓	✓	·	·	·
Turn the public address on or off	✓	✓	·	·	·
See a project and its files	✓	✓	own projects	·	·
Add a photo to a project	✓	✓	own projects	·	·
Remove a photo	✓	✓	own photos	·	·
Move a photo to another print	✓	✓	own photos	·	·
Take a file out of the queue of a printer	✓	✓	·	·	·
Add a print or a live tile	✓	·	·	·	·
Remove a print	✓	✓	·	·	·
Rename a print, mark it Delivered or Discarded, set its source	·	✓	·	·	·
Write the notes of a project or a print	·	✓	own projects	·	·
See the plans and the builds	·	✓	own projects	·	·
Set the GitHub username of a member	✓	✓	·	·	·

Action	Deck	Owner	Member	Share owner	Viewer
Link a repository, slice, approve, reject or send a plate	✓	✓	·	·	·

"Own projects" means the projects of the passkey of the member that still have the member. The API refuses a member session for another project with 401. "Own" photos are the photos that the member added. A deck gets the rights of the owner in the project API, so `ownerOnly` also accepts a deck. The routes that show a plan, a build or the notes refuse a deck.

3 Passkey flows

People sign in with passkeys (WebAuthn). The service keeps no passwords and no email addresses.

3.1 The settings of the passkeys

- **Relying party ID:** `printdeck.beetlebug.org`, the host name of the service. The browser makes and uses each passkey for this domain only.
- **Origin:** `https://printdeck.beetlebug.org`. The API refuses a credential that a page of another origin made.
- **Discoverable keys:** registration requires a resident key. So the sign-in needs no user name: the authenticator gives the credential ID and the user handle.
- **User verification:** "preferred". The authenticator asks for a PIN or a biometric when it can.
- **User handles:** `owner` for the owner, and `m:<project>:<member>` for a member, from the project of its first invite.

Each ceremony has two requests: *begin* gives the options and a sealed state, and *finish* sends the state back with the credential. The API keeps no state between the two (section 3.6).

3.2 Owner enrollment

An enrolled deck asks for an owner code with a signed request. The code works for 10 minutes. The deck shows the page `/owner#code=<code>` as a QR code. The owner scans it with a phone and makes a passkey. Begin only reads the code. Finish deletes the code, and then stores the passkey. Begin also lists the passkeys of the owner as exclusions, so one authenticator does not get a second owner passkey.

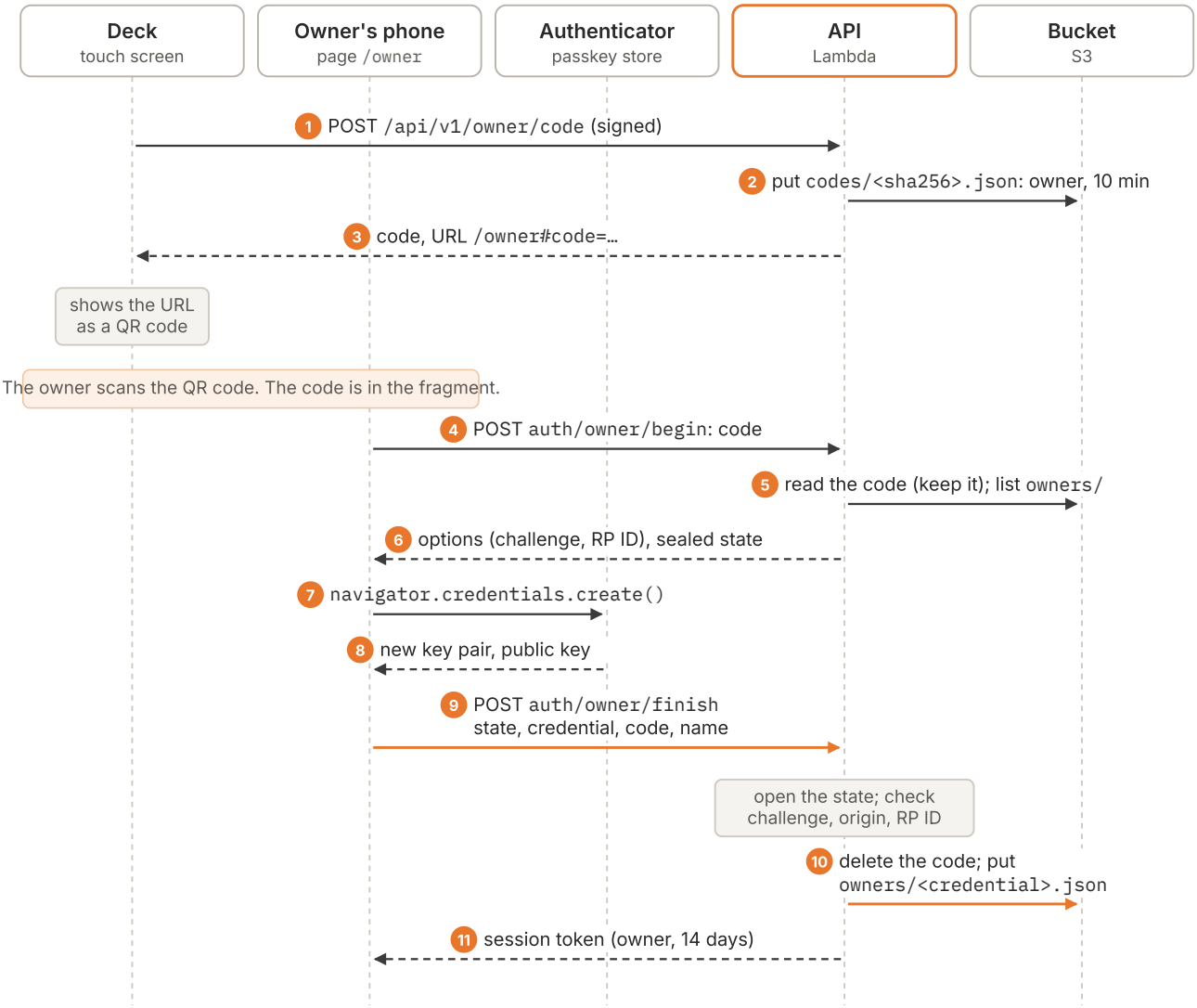


Figure 11. Owner enrollment. The orange steps change the stored identity. The owner can repeat this for each device. Each device then has its own passkey file.

3.3 The invite workflow

The owner invites a friend to a project. The friend gets a link and joins with a passkey. A friend with no passkey for Printdeck makes a new passkey with the link. A friend with a passkey from an earlier invite signs in with it, and the API adds the project to that passkey. The code of the link has 16 random bytes (128 bits). The API keeps only its SHA-256. The invite works for 7 days and works once.

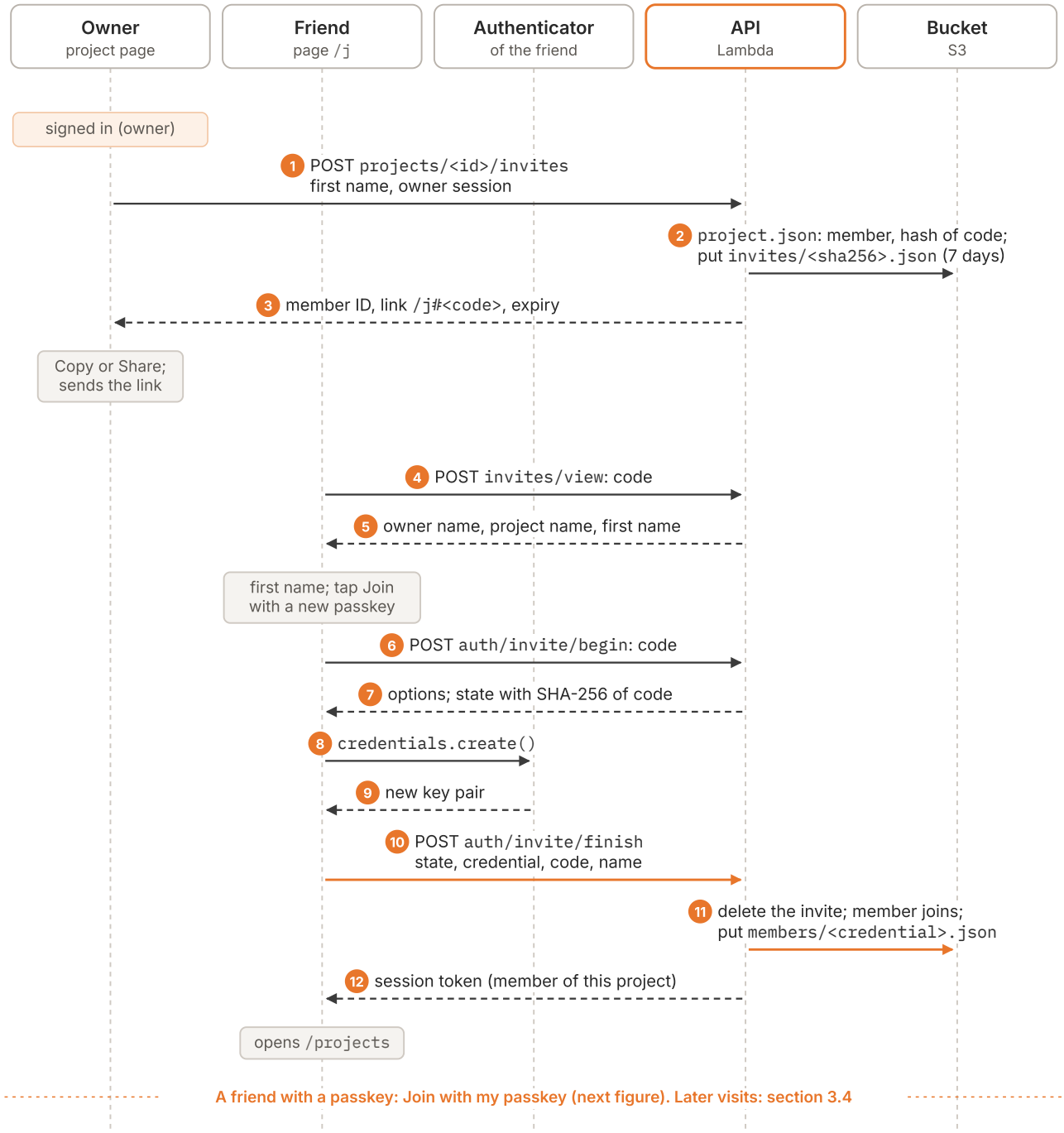


Figure 12. The invite workflow with a new passkey, from the invite to the first session of the friend. The orange steps change the stored identity.

The steps

1. The owner signs in with the owner passkey and opens the project page `/projects/<slug>`.
2. Under Friends, the owner types the first name of the friend and sends the form.
3. The page sends `POST /api/v1/projects/<id>/invites` with the first name and the owner session. Only the owner can do this.
4. The API makes a code and a member ID. It adds a member to `project.json` with the first name, the time of the invite and the SHA-256 of the code. It writes `invites/<sha256>.json` with the project, the member, the name and the expiry (7 days). A project has at most 50 members and invites.
5. The API replies with the member ID, the link `https://printdeck.beetlebug.org/j#<code>` and the expiry. The page shows the link with Copy, and with Share when the browser has a share sheet.

6. The owner sends the link to the friend on any channel.
7. The friend opens the link. The page reads the code from the fragment and sends `POST /api/v1/invites/view`. The reply has the name of the owner, the name of the project and the first name. The page shows "<owner> invited you to <project>", with the first name in a field that the friend can change.
8. The friend taps Join with a new passkey. The page sends `auth/invite/begin` with the code. The API gives the options of a new passkey and a sealed state with the SHA-256 of the code.
9. The browser makes the passkey for `printdeck.beetlebug.org`. The user handle is `m:<project>:<member>`.
10. The page sends `auth/invite/finish` with the state, the credential, the code and the first name. The API checks that the state has the hash of this code, reads the invite again, and checks the credential.
11. The API runs `join`: it deletes the invite file, sets the time of the join on the member, clears the code hash, and adds the passkey file to the member. It writes `members/<credential>.json` with the user handle and a list of projects with this one project.
12. The API replies with a session token for a member of this project. The page keeps it and opens `/projects`, where the friend sees the project.
13. For later visits, the friend signs in with `login/begin` and `login/finish` (section 3.4).

Join with my passkey

A friend with a passkey from an earlier invite taps Join with my passkey. This is a sign-in with the invite code. The page shows this choice first when the tab has a member session. The passkey keeps its user handle from the first invite.

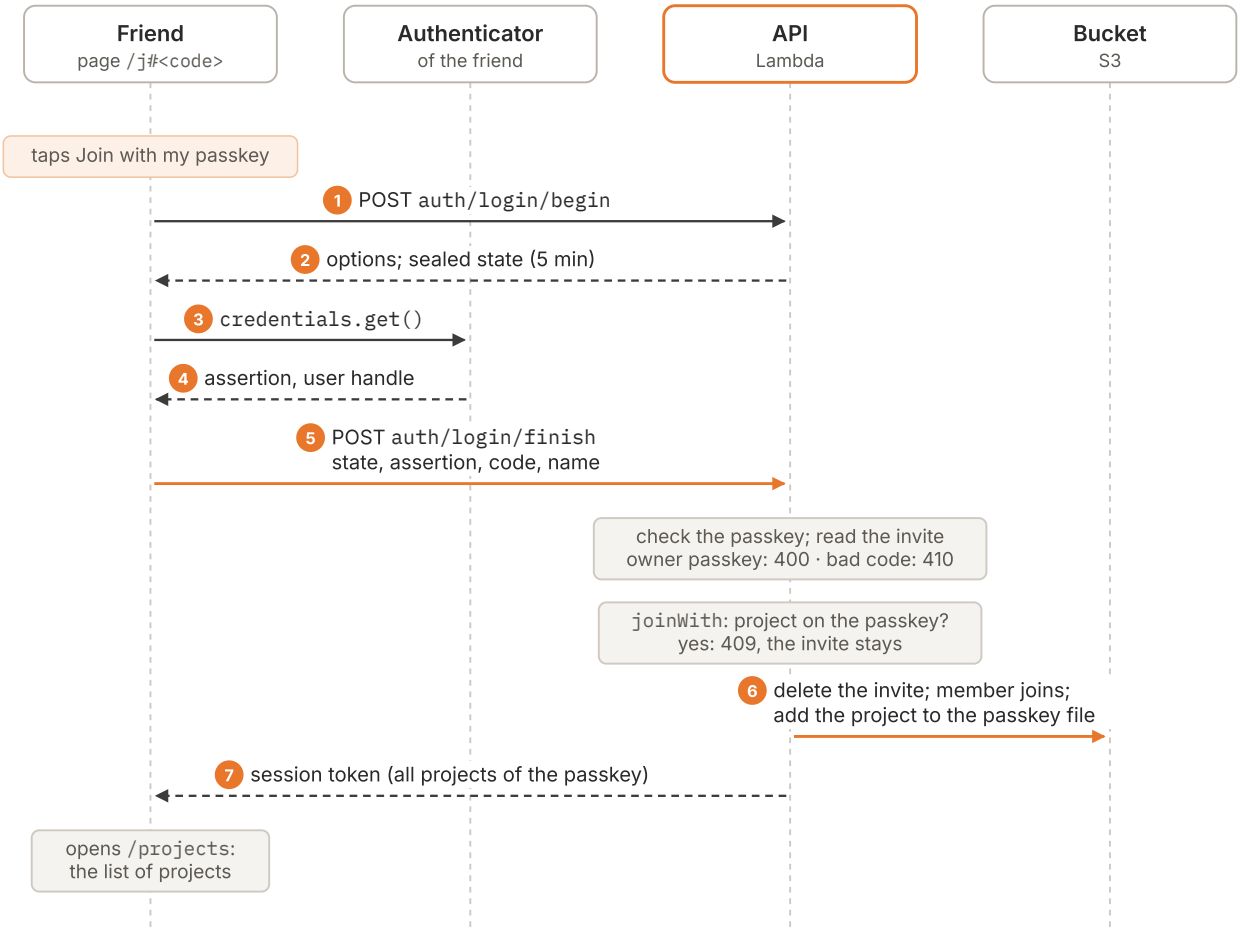


Figure 13. Join with my passkey. The sign-in adds the project of the invite to the passkey. The orange steps change the stored identity.

1. The page sends `auth/login/begin`. The API gives the options of a sign-in and a sealed state.
2. The browser calls `navigator.credentials.get()`. The friend picks the passkey for `printdeck.beetlebug.org`.
3. The page sends `auth/login/finish` with the state, the assertion, the invite code and the first name.
4. The API checks the passkey as in each sign-in (section 3.4), and stores the new counter.
5. An owner passkey with an invite code gets 400. The API reads the invite. A code that is used, removed or expired gets 410.
6. The API runs `joinWith`. A passkey that has the project already gets 409, and the invite stays.
7. Otherwise, `joinWith` deletes the invite file, sets the time of the join on the member, and clears the code hash. It adds the passkey file to the member, and adds the project to the list of the passkey.
8. The API replies with a session token for all the projects of the passkey. The page opens `/projects`, where the friend sees the list.

What the owner sees

The project page lists each member for the owner. A member that did not join yet shows "invited" and the day of the invite. A member that joined shows "joined". Each row has Remove. Remove on an open invite deletes the invite file, so the link stops working. Remove on a member deletes the member from `project.json`. Then `leaveKey` removes the project from each passkey of the member. It deletes a passkey file only when no project is left on it. The next request of the member on this project gets 401. The other projects of the passkey still work. A delete of a project does the same for each member of the project.

A code that does not work

The API replies 410 to a code that is used, removed or expired, and to a code with a wrong form. `invites/view`, `invite/begin`, `invite/finish` and `login/finish` with a code all check the code. The page then shows "This invite does not work" and asks the friend to ask for a new invite. A lifecycle rule also deletes the invite files after 8 days.

What a member can do

A member can	A member cannot
• See each project of their passkey: the prints, the live tiles, the time lapses and the photos. The files come with the signed cookies of each project, for one hour at a time. • See the names of the members that joined. • Add photos to the project or to a print. • Remove the photos that they added, or move them to another print. • Write the notes of the project and of each print. • See the plans and the builds from GitHub, with no keys to change them.	• See a project that they did not join. The API refuses it with 401. • See the open invites of the project. • Rename or delete the project, or change its public address. • Invite friends or remove members. • Remove prints, the photos of other people, or a file from the queue of a printer. • Rename a print, mark it Test, Ready, Delivered or Discarded, or set the link to its source. • Link a repository, slice a plan, or approve or reject a plate.

The API reads the session into a `who` value: owner, or member. `whoFor` finds the member ID of the session in the project of the request. Then it loads the project and checks that the project still has that member. `ownerOnly` guards each owner action. `viewProject` gives a member only the members that joined.

3.4 Sign-in

The owner and the members sign in on the same page with the same two requests. The API finds the passkey file from the user handle and the credential ID. The user handle in the assertion must match the handle of the file. The library checks the signature with the stored public key, and checks the challenge, the origin, the RP ID and the flags. The API then stores the new signature counter.

A member passkey has a list of projects. The session of a member has all of them, each with the member ID in that project. The reply names the project only when the passkey has one project. The page then opens that project. With more projects, the page shows the list. `GET /api/v1/projects` lists each project of the session that still has the member.

A passkey file from before commit e1fcfca has one `project` and one `member`. `Passkey.normalize` reads them as a list of one project. Each sign-in writes the file for the counter, so the file then has the list.

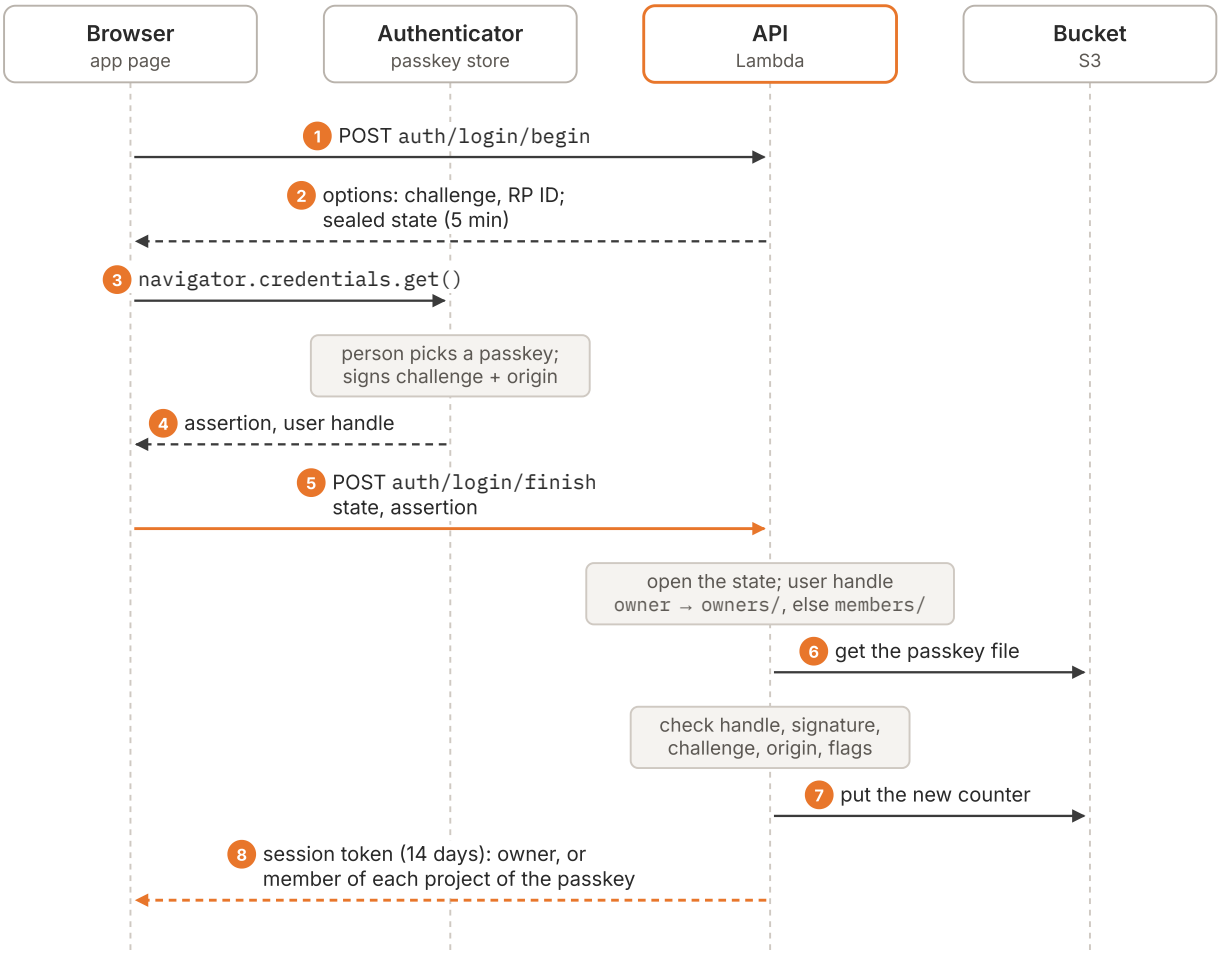


Figure 14. Sign-in. A passkey with no file fails with 401: an owner passkey that the operator deleted, or a member passkey with no project left.

3.5 The session token

A sign-in, an owner enrollment and a join each give a session token. The token is the JSON of the claims and an HMAC-SHA-256 of them. The key is `keys/session`: 32 random bytes that the API makes at the first use. The page keeps the token and its expiry in `localStorage`, so the sign-in stays after the tab closes. A 401 also removes it. The page sends it in the header `Authorization: Bearer <token>`. Sign out clears the stored token on the page.

The claims of a member have `ms`, a map from each project to the member ID in it. A token from before commit e1fcfca has one project in `p` and `m`. The API still accepts it until it expires, after at most 12 hours. `session` loads no project. `whoFor` checks the member in the project of each request, so a removal works at once.

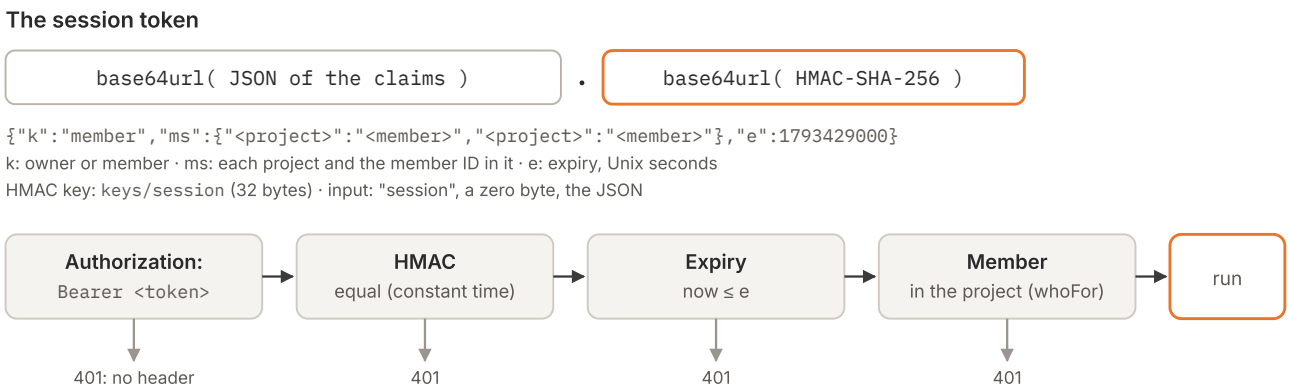


Figure 15. The session token and the checks of each request (`Server.session`, then `whoFor` for a request on a project). A session works for 14 days.

3.6 Sealed ceremony state

The WebAuthn library gives session data at begin: the challenge and the user. The API seals it with the same HMAC key and gives it to the page. At finish, the API opens it and checks the purpose and the expiry of 5 minutes. A page cannot change the challenge, because the HMAC then fails. So any instance of the function can finish a ceremony that another instance began, and the bucket keeps no ceremony files.

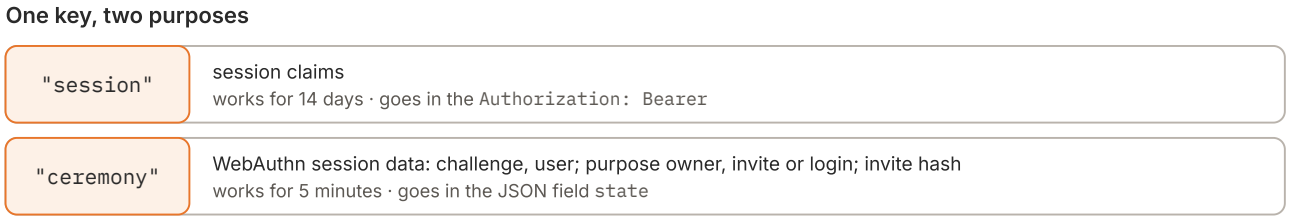


Figure 16. Two kinds of sealed token with one key.

4 Why it is secure

Each row names one threat, the control that stops it, and the place in the code. Section 4.2 gives the limits.

4.1 The controls

Threat	Control	Where
A phishing site gets a sign-in	A passkey works only for the RP ID `printdeck.beetlebug.org`. The browser puts the true origin in the signed client data. The API refuses another origin. A site at another domain cannot get a valid assertion.	`auth.go` `webAuthn()` : `RPID`, `RPOrigins`
A leak of the stored credentials	The bucket keeps public keys of passkeys and decks. It keeps no password, no email address and no private key of a person or a deck. A public key cannot sign in.	`Passkey`, `Deck`
A guessed or reused code	Enrollment codes work for 1 hour, owner codes for 10 minutes, invites for 7 days. Each works once: the API deletes it at use. Codes have about 59 bits, invites 128 bits.	`useCode`, `join`, `joinWith`, `CodeTTL`, `OwnerCodeTTL`, `InviteTTL`
A read of the code files	The bucket keeps only the SHA-256 of each code and invite, as the name of its file. Lifecycle rules delete `codes/` after 1 day and `invites/` after 8 days as a second guard.	`codeKey`, `inviteHash`, `template.yaml`
A changed ceremony state	The state has an HMAC with the purpose "ceremony", an expiry of 5 minutes, and the purpose of the flow. The state of an invite has the hash of its code.	`sealCeremony`, `openCeremony`
A forged session	The token has an HMAC with the purpose "session" and a key that only the API reads. A ceremony state does not open as a session.	`seal`, `open`, `session`
A long-lived session	A session works for 14 days. The page keeps it in `localStorage`, so it stays after the tab closes. Sign out and a 401 remove it. A member session fails at once on a project when the owner removes the member from it.	`SessionTTL`, `whoFor`, `sessionSet`
Cross-site requests	The token goes in the `Authorization` header. The API reads no cookies, so a form on another site sends no session. The file cookies open only the files of one project under `/x/`, and they are HttpOnly and SameSite=Lax.	`session`, `app.go`, `filecookie.go`
A copied authenticator	The API stores the signature counter after each sign-in. The library marks a clone warning when the counter does not go up.	`login/finish`
A member reads too much	A member session has the projects of the passkey. For each request on a project, the API finds the member in that project and checks that the project still has the member. Owner actions need an owner. A member removes only their own photos.	`whoFor`, `lookupName`, `ownerOnly`, `deletePhoto`

Threat	Control	Where
Script injection on the pages	The CSP allows scripts only from the site and WebAssembly. Pages cannot be framed. The headers add HSTS (1 year), `nosniff`, `DENY` frames and no referrer.	`template.yaml` `Headers`
A large or wrong upload	Grants limit the key prefix, the size and the time. The API body limit is 64 KB. Photos are decoded and written again with no metadata: no EXIF, no GPS position and no comment. A test checks it.	`S3Granter.Grant`, `maxBody`, `CleanPhoto`, `TestCleanPhotoStripsMetadata`
A forged deck request	Each request has an Ed25519 signature of the method, path, time, request ID and body hash. The time must be within 5 minutes.	`cloud.Verify`, `Server.deck`
A guessed address	Share and project IDs have 13 random bytes. A missing share or file shows the same "not found" page.	`newID`, `CustomErrorResponses`
A person reads the files of another project	CloudFront serves `/x/*` only with a valid signed cookie. The policy of a cookie names one project, and the cookie path is that project. The API signs cookies only for projects that the session can see. The files have the same origin as the pages, so reads need no CORS, and CloudFront can keep them in its edge cache.	`filecookie.go`, `template.yaml` `/x/*`
A forged token of an Action	The API checks the RS256 signature with the keys of GitHub (JWKS), the issuer `https://token.actions.githubusercontent.com`, the audience (the host name of the service) and the expiry, with 1 minute of leeway. The token must have the claim `repository`.	`GitHubOIDC.Verify`, `buildClaims`
A token used twice	The API writes `oidc/<sha256>` of the `jti` with a conditional write. A second use gets 401. The Action asks for a new token for each request.	`useToken`, `tokenKey`, `template.yaml` `oidc-tokens`
A repository writes to another project	The repository of the token selects the project through `repos/<owner>/<name>.json`, and the project must still name it. Only the owner sets the link. A plan or a build of another repository gets 403.	`projectOfRepo`, `setRepo`, `actionBuild`, `planDone`
A changed workflow fills a build	Each request on `/api/v1/builds/` needs `job_workflow_ref` of `.github/workflows/printdeck.yml` on main or master. A copy on a branch or in a pull request gets 403. It can only send a plan.	`sliceRun`, `SliceWorkflow`
A fork sends a plan	The token of a fork names the fork. The owner of the fork must be a member who joined, with the GitHub username that the owner set. This check needs no call to GitHub. Then the App must find an open pull request into the repository of the project, from that branch, with the commit of the token as its head. A fork never fills a build.	`forkProject`, `OpenPull`, `planDone`
A profile runs code in the slice run	`curaslice.py` parses each formula of a profile, and checks each node of its syntax tree against a list. A formula can use only constants, names, operators, the functions of Cura and `math`, and `index` or `count` on a list of constants. A name or a string with `__` fails. The formula then runs with no built-ins.	`check_formula`, `safe_attribute`

Threat	Control	Where
A large upload of the Action	The grant of a plan or a build has one folder, 200 MB for each file, and 1 hour. Done checks that each named file is in the folder. Each file of a build has its size and SHA-256 in the record.	`MaxBuildFile`, `newPlan`, `buildFiles`, `buildDone`
A member slices or approves	Only the owner makes a build, and approves or rejects a plate. A member sees the plans and the builds with no keys.	`newBuild`, `decidePlate`, `ownerOnly`
A leak of the token of the App	Each installation token works for one repository, with only the permissions of its call: Actions (write) to start a slice run, Pull requests (read) to find a pull request or read its state, or Pull requests (write) to write the comment of Printdeck. Each also has Metadata (read). The App JWT ends after 9 minutes. The private key is in Secrets Manager.	`installationToken`, `Dispatch`, `OpenPull`, `PullState`, `Comment`
A bad link to a source	The source of a print must be an `http` or `https` address with a host, of at most 500 characters. Only the owner sets it.	`editPrint`
A public bucket	The bucket blocks public access and encrypts at rest. CloudFront reads only the listed keys with OAC. It reads the print files, the photo pictures, the live pictures, the plan models and the build files under `x/` only when the stack has the key of the file cookies, and never the JSON records. It also reads `gh/`, the public pictures of the comments on pull requests. The function has get, put, delete and list on this bucket only. It can also invoke itself, for the jobs after a request.	`template.yaml`
The app or the queue starts a resin print	The API of the resin deck refuses uploads, each start and each resume with 409. Its printer for the API does nothing on PrintFile and Resume. A start from ChiTuBox during a print goes to the queue, and with files in the queue the deck asks on its screen. Only a press on the touch screen sends M6030 or a resume.	`octoapi.Server.Resin`, `resinAPIPrinter`, `ChiTu.RequestStart`
A project sees another project's prints	The service keeps the queued files of each project in the project (`x/<project>/queue/`). The state of each printer (`printers/<id>-live.json`) goes only to `GET printers`, which only the owner and the decks get.	`setQueue`, `projectQueued`, `routePrinters`
A changed file of a plate	The deck checks the size and the SHA-256 of a plate file against the build record before it adds the file to its queue.	`downloadTo`
A deck writes for another deck	The queue, the removals from the queue, the state and the plates of a printer need a request of the deck of that printer.	`deckPrinter`, `setQueue`, `queueRemoves`, `removesDone`
A changed update file	A deck installs only a bundle with an Ed25519 signature of `SHA256SUMS` by the release key. The debug image also accepts an unsigned bundle.	`deck/internal/system/sign.go`

The CSP of the pages

`default-src 'self'; script-src 'self' 'wasm-unsafe-eval'; style-src 'self' https://fonts.googleapis.com; font-src https://fonts.gstatic.com; img-src 'self' data: blob: https://<bucket>; connect-src 'self' https://<bucket>; frame-ancestors 'none'; base-uri 'none'; form-action 'self'`

The pages have no inline scripts. `wasm-unsafe-eval` lets the page compile its own WebAssembly. `connect-src` allows the bucket, for the upload grants, and for the signed S3 links of a stack with no cookie key. The files of the projects come from `'self'`.

4.2 The limits

These limits come from the code as it is. Each one has a reason or a way to act.

- **A stolen session token works until it expires.** The API keeps no list of sessions. A token works for up to 14 days. The browser keeps it in `localStorage` after the tab closes, so a person who can use that browser profile can use the session. Sign out clears the token only on the page. To end all sessions, replace `keys/session`. A running instance of the function keeps the old key in memory until it stops.
- **The device of the owner holds the owner passkey.** A person who can unlock that device, or the account that syncs its passkeys, can sign in as the owner. User verification is "preferred", so an authenticator with no PIN or biometric can also sign in.
- **An owner passkey has no remove action in the API.** To remove one, the operator deletes its file in `owners/`.
- **A deck holds owner rights.** A signed deck request has the rights of the owner in the project API. A deck can also ask for an owner code. A person with the SD card of a deck has its key. A person who can use the touch screen, or the remote screen of the deck, can make an owner code. Use `pwf decks remove` after the loss of a card.
- **The counter is a signal.** The API stores the counter and the clone warning of the library, but it does not refuse a sign-in with the warning. Synced passkeys often report a counter of 0, and then the check does not apply.
- **A ceremony state can be sent again.** The API keeps no list of used challenges. The same state and assertion can give a second session within the 5 minutes of the state. TLS protects both.
- **The replay check of deck requests is partial.** Each function instance keeps its own list of request IDs. The signature, the 5-minute window and TLS remain.
- **Links are bearer secrets.** A person with a share link sees the print until the share ends. A person with an invite link can join in place of the friend, until somebody uses it or it expires. A person with an invite link and their own member passkey can join with that passkey. The owner can remove a member that joined in that way.
- **A member passkey opens all its projects.** A person who can unlock the device of a friend sees each project that the friend joined. To end that access, the owner removes the member from each project.
- **The AWS credentials of the owner are above all other keys.** `pwf` uses them on the computer of the owner. A person with them can make enrollment codes, read `keys/session` and change the stack. Keep that computer and its AWS profile safe.
- **A file cookie works for up to one hour.** A member that the owner removes keeps the cookies of the project until they expire. The API refuses the member at once, but CloudFront checks only the signature and the time.
- **Fixed: a file cookie opened the JSON records.** Before commit 71fae88, the bucket policy let CloudFront read all of `x/*`. A member with the cookie of a project could read `project.json`, with the names of the open invites and the SHA-256 of their codes. Now CloudFront reads only the print files, the photo pictures, the live pictures, the plan files and the build files. The JSON records stay with the API.

- **The key of the file cookies is in Secrets Manager.** The role of the function and the AWS credentials of the owner can read it. A person with it can sign cookies for any project.
- **Share links are public.** Section 2.5 explains this choice and its protections.
- **The bucket is the root of trust.** `keys/session` is in the bucket. A person who can read the bucket can make session tokens. Only the operator and the function have access.
- **The function URL has no AWS auth.** A request can go to the Lambda URL past CloudFront. The API checks the auth of each request in the same way. The CloudFront headers do not apply there, and the API replies only JSON.
- **The remote screen of a deck uses plain HTTP.** Its password protects it on the home network only.
- **A deck holds owner rights for the builds too.** A signed deck request can link a repository, slice a plan, and approve or reject a plate.
- **The slice run reads the files of the plan commit.** A plan can come from the head commit of a pull request, also from a fork. The slice run checks out that commit and reads its `printdeck.yml` and `profiles/`. Each formula of a profile passes the check of its syntax tree (section 4.1), and then runs with Python `eval` and no built-ins. The check is a list of the parts of Python that Cura uses. It is not a sandbox of the process. Look at the changes to `profiles/` before a slice of a pull request.
- **The comments on pull requests make pictures public.** The comment of Printdeck shows the changes of the models, the plates and the first photo of each print of a plate. The service copies these pictures to `gh/` in the bucket, with names of 16 random bytes, and anybody can read them. The comment also has the links to the plan and the build pages. On a public repository, anybody can see the comment.
- **The models of a fork come from the run of the fork.** The `generate` step of a fork runs in the fork, so its STL files are not proof of the CAD source. Look at each part in the 3D view of the plan page, and at each plate in the preview of the build page, before an approval.
- **A member's GitHub username is the key of a fork.** The owner types it. A wrong username lets that GitHub user send the plans of their open pull requests. Check each username.
- **A slice token works for each build of its repository.** The API checks the repository of a build only. The spec also asks for a check of the commit.
- **The workflow rule names main and master.** `sliceRun` accepts only these two branches. The App starts the workflow on the default branch of the repository. A repository with another default branch gets 403 on each request of its slice run.
- **Each workflow of the repository can send a plan.** A copy of the workflow on any branch, with `id-token: write`, can send a plan. The plan page shows its parts and models to the owner and the members. A fork can send a plan only for an open pull request of a member.
- **The slice run needs the GitHub App.** When the stack has no secret `<stack>-github-app`, the service cannot start a slice run. The build page then shows the owner a `gh workflow run` command to run by hand.
- **The estimates of the queue are rough.** A start time is the time left of the print now, the print times of the slicer of the files before it, and 10 minutes for each change of a part.
- **The Apple team ID is public.** The association file of the app and each signed app have it. It is not a key.
- **Plans and builds have no expiry.** They stay until the owner deletes the project. The spec gives 30 days for test builds and 1 GB for a build, and the code has neither.

5 The data of a print

A deck sends a print to up to three places: the live tile of its project, the share page, and the project record at the end.

5.1 An FDM print on printdeck

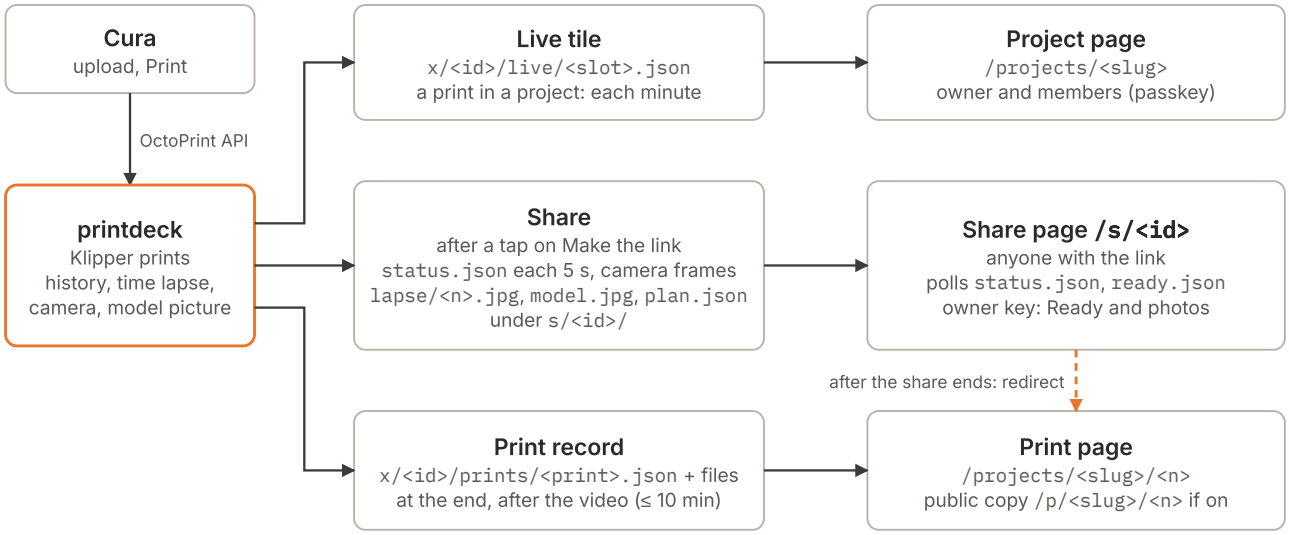


Figure 17. The data of an FDM print. Each arrow into the middle column is a signed request and uploads with a grant.

1. **Upload.** Cura sends the G-code through the OctoPrint API of the deck. The Printdeck plugin can also pick the project of the next prints.
2. **Live tile.** For a print in a project, the deck looks at the state each 15 seconds. It sends the progress each minute, and at once at the start, at a pause and at the end. The API keeps one record for each printer: `x/<project>/live/<slot>.json`. The project page hides a tile with no update for 5 minutes.
3. **Share page.** After a tap on Make the link, the deck uploads `status.json` every 5 seconds with the newest camera frames. It also uploads each new time lapse frame, the picture of the model (at most each 15 seconds), and `plan.json` once. The page polls `status.json` and `ready.json`. The status files have a cache time of 2 to 5 seconds.
4. **End of the share.** At the end of the print, the deck tells the API. The share stays for 24 hours.
5. **Project record.** The deck waits for the time lapse video, for at most 10 minutes. Then it sends the print record, the picture of the model, the plan and the time lapse frames to `x/<project>/prints/<print>/`. The print gets the next number in the project. The photos of the share go to the print.
6. **Move.** When the share ends, the sweep deletes its files. Its page then sends visitors to the print page `/projects/<slug>/<n>`, or to the public page when the project has one.

5.2 A resin print on resindeck

The flow to the cloud is the same. The source of the data is different. The ChiTu board reports the bytes of the file that it read. So the deck reads its own copy of the sliced file for the layers and the pictures. An approved resin plate of a build also goes into these copies, and the deck then sends it to the board.

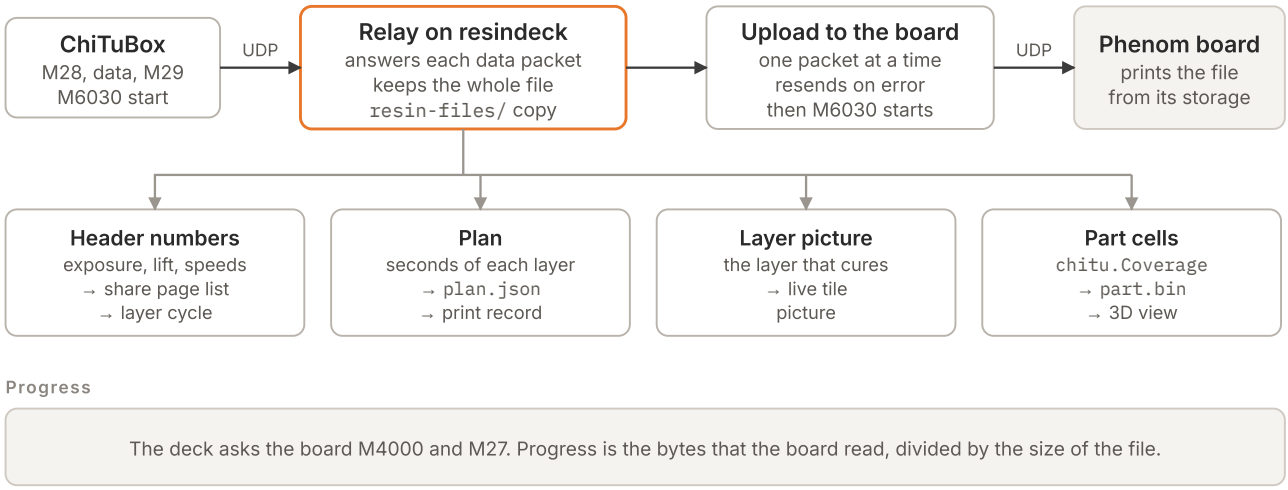


Figure 18. The data of a resin print. The relay keeps a copy of each upload, and the copy gives the numbers, the plan, the layer pictures and the cells.

- **Numbers.** The header of the file has the exposure, the bottom layers, the lift and the speeds. The share page lists each number, and it plays the cycle of one layer in step with the printer: light, lift, return and rest.
- **Live tile.** The picture on the tile is the layer that the printer cures now. The deck uploads a new picture when the layer changes.
- **3D view.** The deck makes `part.bin` once for a share, and the share page draws the part in 3D.
- **Restart of the deck.** A resin print continues through a restart of resindeck, because the board prints the file. The share continues too: the deck saves the share to `share-resume.json` in its state folder every 30 seconds, and loads it after the start when it has no share.
- **One tile.** When the live tile of a deck names its share, an older open share of the same deck shows no tile on the project page. An open share with no new status for 30 minutes also shows no tile. A deck that lost the share at a restart never ends it.

The cells of `part.bin`

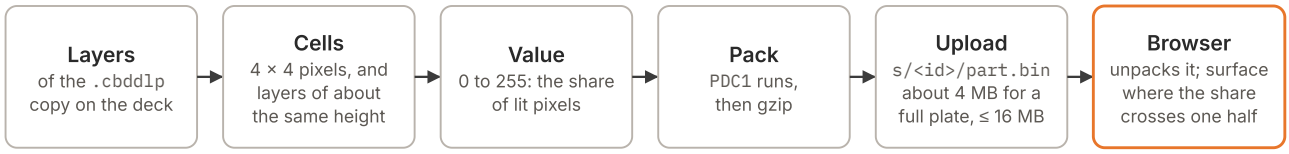


Figure 19. From the sliced file to the 3D view. Format `PDC1`: the cell size in mm, the slice height, W, H and D, then for each slice the runs of a value and a count.

A cell is 4 × 4 pixels of the screen, and a stack of layers of about the same height. Its value is the share of its pixels in all its layers that are lit, from 0 to 255. The page puts the surface where the share crosses one half. So an edge sits at its true place inside the cell, and the view is as fine as the slices. The file is the mirror of the screen for a printer with projector 1, as the slicer shows the part. A full plate gives about 4 MB. When the upload of `part.bin` fails, the share continues, and the deck tries again after 10 minutes. `part.bin` stays with the share only. The project record has no 3D cells.

5.3 The print page: names, notes and delivery

People change some fields of a print on the pages. These fields stay when the deck sends the print again (`Title` and `pageFields` of `PrintRecord`). Each change uses `PATCH projects/<id>/prints/<print>`. A deck gets 401 on this route.

- **Rename.** The owner can give a print a title of up to 120 characters. The pages show the title in place of the name from the deck, and the name from the deck under it. A title that is the same as the name clears the title.
- **Notes.** The owner and the members write notes on a print, and on the project with `PATCH projects/<id>/notes`. Notes have at most 4000 characters. The record keeps the name of the last writer and the time.
- **Test print.** The owner marks a print as a test print: a print to try a setting or a fit, and not to give. The API records the time. A test print is not in the card Ready for pickup.
- **Ready for pickup.** The owner marks a print Ready for pickup on its page, before Delivered, and the API records the time.
- **Delivered.** The owner marks a print Delivered, and the API records the time. The owner can also clear the mark.
- **Discarded.** The owner marks a print Discarded when it is not good enough, with a note of why, of up to 500 characters. The API records the time. A second tap clears the mark and the note. A discarded print is not in the card Ready for pickup. The iPhone app can also set the mark.
- **Source.** The owner sets a link to the source of the model, for example a file of a commit on GitHub. The link must be an `http` or `https` address with a host, of at most 500 characters.
- **Ready from a share.** When the owner of a share marks it Ready, the API copies the time and the note to the print of the share in its project. A clear of Ready also clears the mark on the print. The API also copies the mark when a share joins a project later.

The print page shows "Discarded" with the day and the note, else "Delivered" with the day. Before the delivery, it shows the Ready mark when the print has one. A tile has the badge Discarded, Test, Ready to pick up, or Delivered with the date. A member can change the notes only. The other fields get 401 for a member.

Photos from a phone. The photos of a project and of a print have Take photos on a phone: the camera opens, and after each photo goes up, Take another photo opens it again. A photo that does not go up does not stop the others.

Move a photo. A photo can move to another print of the project, or to no print, with `PUT projects/<id>/photos/<photo>`. The owner moves any photo, and a member moves only their own.

5.4 The photo viewer

`assets/lightbox.js` shows a photo at full size over the page. The app pages, the share page and the public pages use it. A link with `data-photo` in a photo grid opens the viewer.

- Previous and Next go through the photos of the same grid. The arrow keys do the same, and Escape closes the viewer.
- Each photo has an address: the address of the page with `#photo=<id>`. Copy link copies it. An open of that address shows the page with the photo over it.
- A tap on a photo adds its address to the history of the browser, so Back closes the viewer. Full size opens the file in a new tab.

The fragment does not go to the server. A person with the address of a photo also needs access to the page: a passkey of the project, the share link, or a public page.

5.5 The project page

The page `/projects/<slug>` is the home of a project. Its head shows the name, the start of the project, the circles of its people, and links to the repository and the public page. Tabs lead to the other parts. Each tab has its own address. A tap changes the tab with no new load of the project.

Tab	Address	What it shows
Overview	`/projects/<slug>`	What happens now, the numbers, the chart, the newest prints, the feed, and a column of cards (Figure 20).
Prints	`/projects/<slug>/prints`	All the prints, with a chip for each printer, and chips for the prints ready for pickup, the delivered prints (the newest delivery first), the test prints and the discarded prints.
Photos	`/projects/<slug>/photos`	All the photos, and Add photos. Each photo has Move to, which puts it on another print or on no print.
Builds	`/projects/<slug>/plans`	The plans of the pull requests and of the tags (releases), and their builds. The Overview shows the newest plan of main. The tab shows only for a project with a repository.
Settings	`/projects/<slug>/settings`	The repository, the printers, the friends with their GitHub usernames, the public address, and Delete.

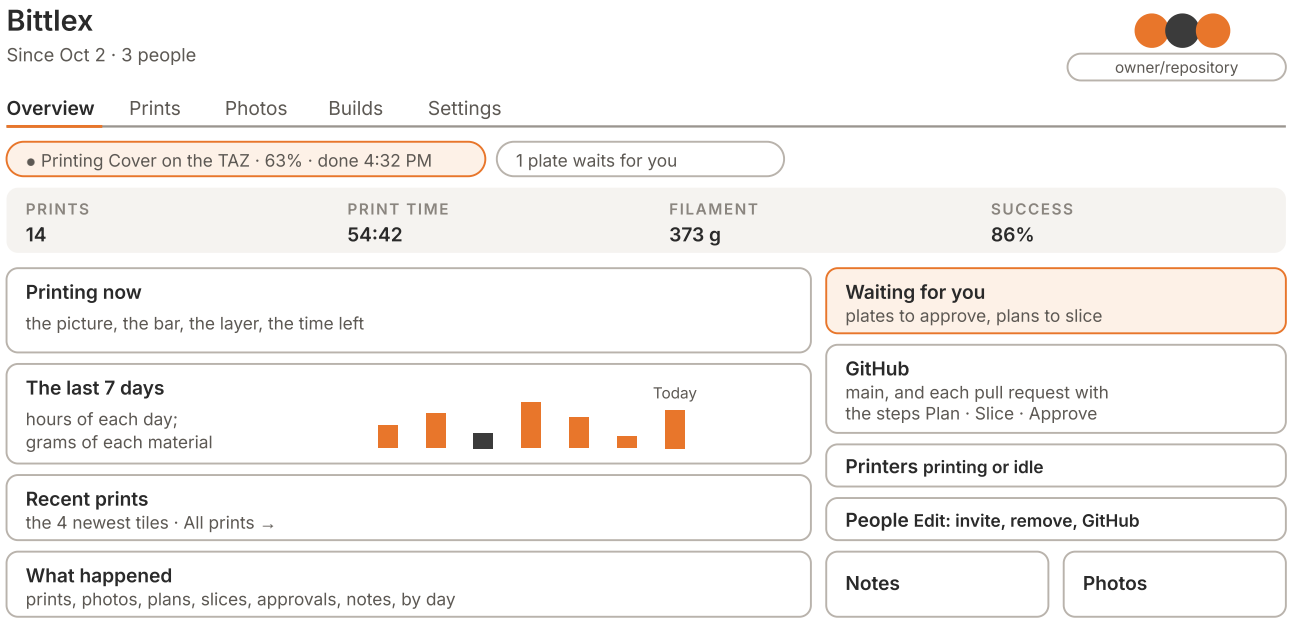


Figure 20. The overview of the project page on a wide screen. On a narrow screen the cards make one column, and Waiting for you and Printing now come first.

- **The line of what happens now** names the print that prints, with its share and the time of its end, or the time of the last print. It also counts the plates that wait for the owner and the open pull requests. Each item is a link.
- **The numbers** are the prints, the print time, the filament and the share of the prints that finished.
- **The chart** shows the hours of print of each of the last 7 days: the finished prints, with the stopped prints over them. Each bar has the short name of its day, and the last bar is Today. A bar under it shows the grams of each material, in the color of the filament.
- **The feed** shows the 12 newest events, with a heading for each day: the end of a print, Ready, Delivered and Discarded, the photos of one person on one day, the plans, the slices, the approvals, the rejections

and the notes.

- **The GitHub card** shows the newest plan of main and each open pull request, with its author and the state of its three steps: Plan, Slice and Approve.
- **The printers card** shows each printer of the project: printing with its share, paused, or idle with the time of its last print. A deck from before the printers sends its live print with no printer ID. Its row then has the name of the printer from the deck. A printing row shows the percent and the time of the end. A resin printer also shows its layer, for example layer 120 of 800, and a picture of the layer that cures now.
- **The card Ready for pickup** shows each print that waits, with small pictures of its photos (else of its model), the time of the mark, and Delivered for the owner. It links to the delivered prints.
- **The card In the queue** shows the files of the project in the queues of the printers: the printer, the place and the estimate of the start (section 7.4). The owner can take a file out with Remove (`DELETE projects/<id>/queue/<printer>/<item>`), and the deck does it within a minute. The line of what happens now gives the start of the next file.
- **The tiles** of the prints with photos have small pictures of up to three photos in a corner.
- **The People card** shows the owner, the members with their GitHub usernames, and the invites that wait. The owner taps Edit to set a GitHub username, remove a member, cancel an invite, or make an invite.

The page reads the project again each minute, or each 30 seconds while a print prints. The first load is one request, `GET /names/<slug>?view=1`. The reply has the project and its view. `pwf.js` starts this request before the WebAssembly loads. The API reads the records of the project at the same time, 64 at a time. It draws the line of what happens now and the printers at each read. It draws the other cards again only when the prints, the photos, the plans or the notes change, so the pictures do not load again. The pages have the cache rule `no-cache`: after a deploy, the next load gets the new page.

The head of the project. The name, the people, the links and the tabs are over each page of a project: the overview, a print, a printer, the plans, a plan and a build. The tab of the page is marked.

5.6 The 3D view of a print

The deck uploads the G-code of an FDM print with gzip, as `print.gcode.gz`: once for the share (up to 15 MB after gzip), and with the print at its end. The share page and the print page draw it with the G-code viewer of the build page (`viewer3d.js`, `showGcode`) in a compact mode with no controls. A print with no G-code shows the picture of the model from the deck. The view of a TAZ print draws the real bed of the TAZ under the part, from the CAD of LulzBot. Another printer has the plain bed with the grid.

- **The layers.** The layers that printed have the colors of Cura. The later layers are a faint ghost. Each mesh of moves has a twin with the moves in the reverse order, so a step of the print only changes two counts.
- **The nozzle.** A hot end (the tip, the hex and the heater block) at the move that prints now, drawn over the part with an orange edge.
- **The sync with the printer.** The status of the share has `pos`: the byte of the file that Klipper reads. The viewer keeps the byte of each move, puts the nozzle at the move of that byte, and goes on at the speeds of the G-code until the next status. It starts 1.5 s behind, because Klipper reads ahead of the nozzle. With no byte, the viewer uses the time of the layer.
- **The camera view.** The status can also have `view`: a 3 × 4 matrix from the G-code coordinates to the frames of the time lapse. The deck makes it from the calibration of the camera with the bed at the park, the turn of the frames, and the crop to the part (`composeView`). The view then has the shape of the frames, and Turn freely gives the view that turns. The deck sends the view only with a calibration of the towers (`BedY`): a calibration of the nozzle on a printer with a bed that moves in Y has no direction of Y. The page also ignores a matrix that makes the edge of the part in Y shorter than a tenth of its other edges.

On a wide screen the numbers of a share are across the full width, and the model and the time lapse are side by side under them, with the same height. On a phone the numbers come first.

The picture of a record. A print record keeps the size and the modification time of its file at the end of the print. A slicer can send another file under the same name later. The deck draws no picture of a record when its file changed.

6 Builds from GitHub

A project can link one GitHub repository. At each push, the Action sends the parts of the commit. The owner chooses the plates, and the Action slices them.

6.1 The flow

1. **Push.** A person pushes a branch or a tag. The workflow `.github/workflows/printdeck.yml` starts in the repository that gets the push: the repository of the project, or the fork of a member.
2. **Plan run.** The job `plan` reads `printdeck.yml` and runs its `generate` step. It finds the parts that changed. Then it sends the plan, with an STL model of each file of each part and the PDFs of its `docs`.
3. **Plan page.** The owner opens `/projects/<slug>/plans/<id>`, and chooses FDM or resin first. Each part has one row, with a chip for each of its variants, for example a fine and a standard quality. The parts that changed are checked at first, and a release plan checks all the parts. The owner can change the checks, the variants and the copies of each part.
4. **Plates.** The page suggests the plates, and checks that each plate fits on its bed. The owner can move a part to another plate or to a new plate.
5. **Slice.** The owner taps Slice. The API makes a build with the state "slicing". The GitHub App of the service starts the workflow on the default branch, with the input `build`.
6. **Slice run.** The job `slice` gets the plates from the API, and checks out the commit of the plan. It slices each plate, sends the facts of the files, uploads the files, and marks the build done.
7. **Build page.** `/projects/<slug>/builds/<id>` shows each plate with its facts: the profile with its version, the material, the time, the grams or ml, the layers, the file and the slicer. A 3D preview draws the moves of the G-code of a plate, layer by layer, with the colors of Cura. While the build slices, the page shows what the slice run does, and reads the build again each 10 seconds.
8. **Approval.** The owner approves each plate, or rejects it with a reason. The API records the state, the name and the time. The owner then sends an approved plate to a printer: it goes to the queue of the deck (section 7.3).

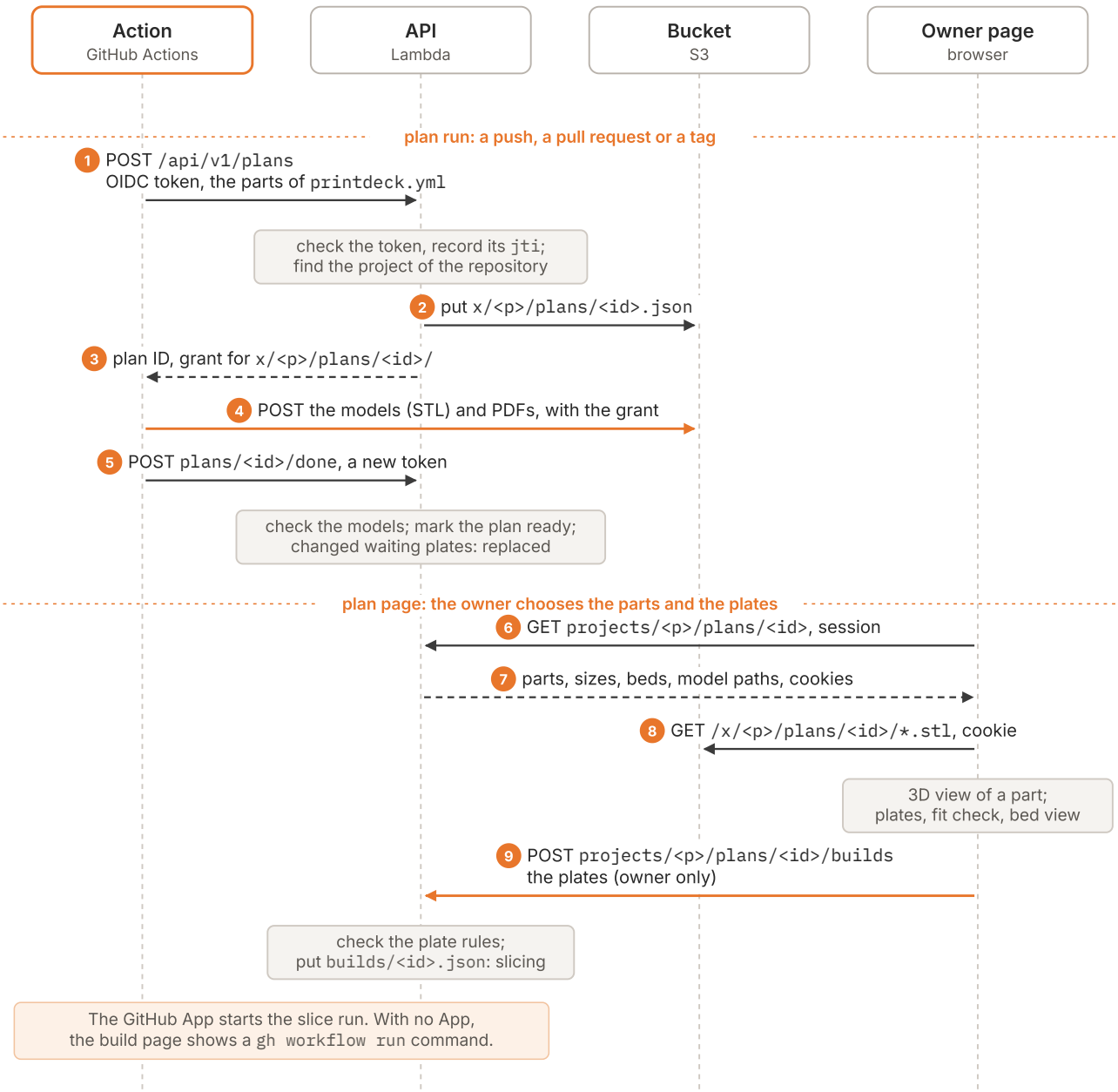


Figure 21. The plan run and the plan page. Each request of the Action has a new OIDC token. The orange steps upload with a grant, or make a build.

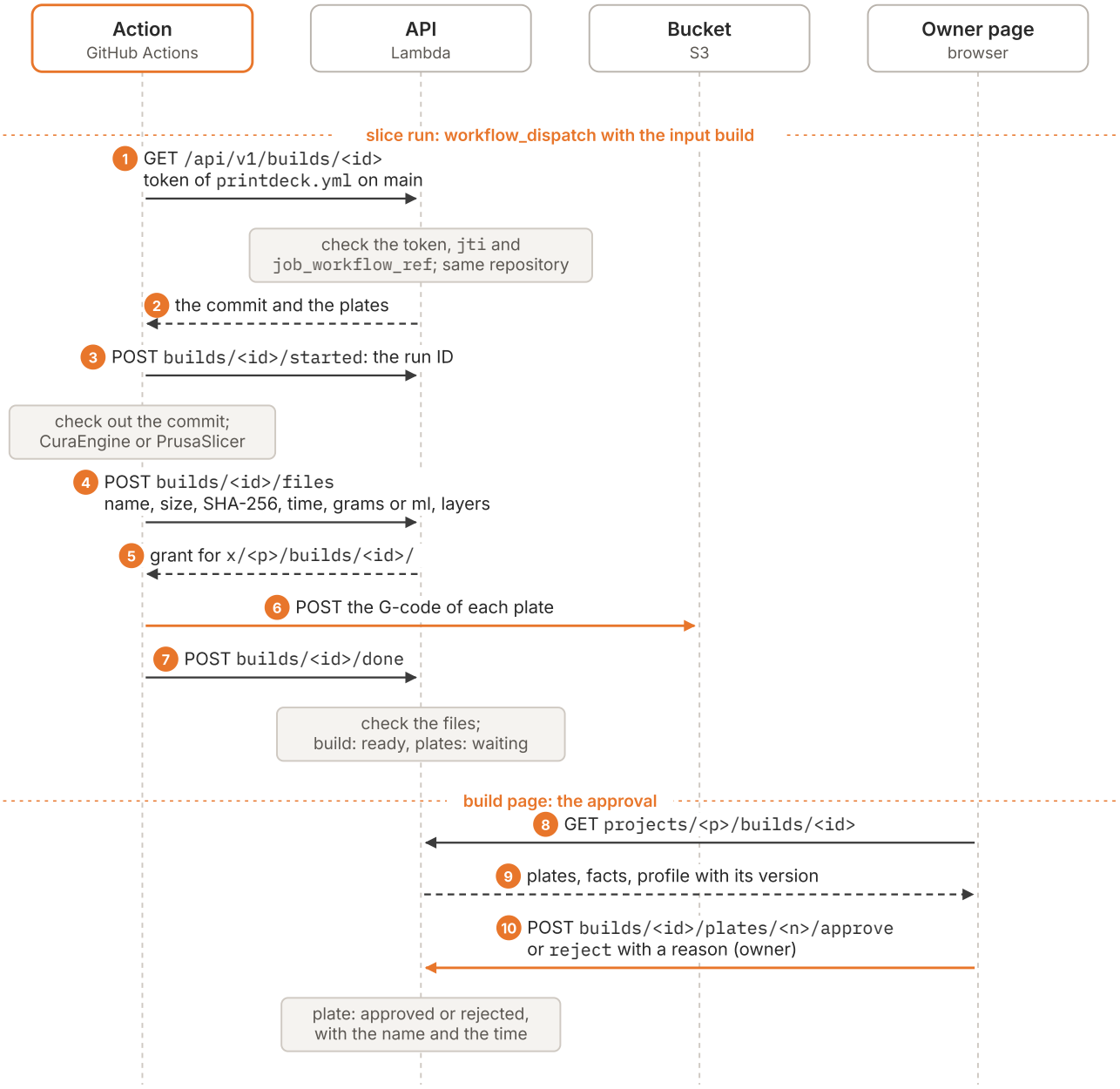


Figure 22. The slice run and the approval. Each request on `/api/v1/builds/` needs a token of the workflow `printdeck.yml` on main or master.

6.2 The plans and the builds

The bucket keeps each plan and each build as a JSON record of the project, with a random ID of 10 bytes. The overview of the project page shows the open pull requests and the newest plan of main (section 5.5). The overview hides a closed or merged pull request. The API asks GitHub with the App for the state of each pull request, at most each 10 minutes for an open one and each 6 hours for a closed one. The Builds tab and the page `/projects/<slug>/plans` show only the plans of pull requests and of tags, for example `v1.2` or `release/1.2`.

Record	Key	What it has
Plan	`x/<project>/plans/<id>.json`	The repository, the commit, the ref, the kind, the pull request, the tag or the branch, the run ID, and "ready". A plan of a fork also has the fork and the GitHub username of its member. For each part: the name, the variant, the files, the count, the printer, the process, the profile with its version, the material, `prebuilt`, `alone`, `changed`, the digest, the model files, the size of the model, the bed and the margin. The plan also has its PDFs (name, file, size, pages, SHA-256), and for a pull request the base plan and the changes of its models.
Plan models	`x/<project>/plans/<id>/part<i>-<j>.stl`	One STL for each file of each part, at the middle, for the 3D views. A 3MF file loses its support blockers here. Each PDF is `doc<n>.pdf`, with a PNG of its first page, `doc<n>.png` (600 pixels wide).
Build	`x/<project>/builds/<id>.json`	The plan, the repository, the commit, the state, the error, the run ID and the plates. Each plate has the printer, the process, the profile, the material and the parts with their counts. After the slice, it has the file, the size, the SHA-256, the time, the grams or ml, the layers, the layer height and the slicer. The approval adds the state, the name, the time and the reason. A plate that the owner sends has the printer and the times of the send and of the take. A plate with a picture for the comment has its public address.
Build files	`x/<project>/builds/<id>/plate-<n>.gcode`	The sliced file of each plate. A resin plate has a `.cbddlp` file, for example `plate-1.cbddlp`: the slice of PrusaSlicer, or the `prebuilt` file.

The kind of a plan comes from the ref of the token. The body of the request cannot change it.

- **test:** a branch with an open pull request. The API asks GitHub with the App: a branch of the repository with an open pull request, with the commit of the token as its head, makes the plan of that pull request. A plan of a member's fork is always of this kind (section 6.8).
- **main:** the branch main or master.
- **branch:** another branch, with no open pull request.
- **release:** a tag, for example `v1.2`.

The commit comes from the token. A plan of a fork must use the commit of its token. It must have 40 hex digits. A plan has 1 to 50 parts. A part has at most 50 files and 1 to 100 copies. A build has 1 to 20 plates.



Figure 23. The states of a build and of its plates. A new run of a failed build sets it back to "slicing". Slice again on the build page makes a new build with the same plates.

When a new plan of a pull request or a branch is done, the API marks the waiting plates of the older plans of the same pull request or branch as "replaced", when a part of the plate is gone or changed. The Action sends a digest of each part: the SHA-256 of its entry, its files and its profile. With no digest, the mark `changed`, the profile and the material decide. The approve and reject routes refuse a replaced plate with 409. They also refuse a plate of a build that is not ready.

6.3 The plates

- The parts of one plate have one printer, one profile and one material. The API compares each part with the first part of the plate, and refuses a mix with 400.
- A part with `alone: true` or a `prebuilt` file prints alone on its plate. The API refuses it on a plate with other parts. A prebuilt part has 1 copy.
- The suggestion puts the checked parts with the same printer, profile and material on one plate while they fit. A part that does not fit gets a new plate.
- The fit check uses the grid of `curaslice.py`: the objects in rows, each with the room for its brim or skirt, and 10 mm from the edges of the bed. The plan has the bed and the room of each part. A plate that does not fit shows how many objects fit, and Slice stays off.
- Each plate has a view of the bed from above, with the objects at the places of the grid. The 3D view shows the models of one part on the bed of its printer.
- One object alone goes to the middle of the bed. The plan page does not compare the `settings:` of the parts. A plate with two values of a plate setting fails in the slice run.

6.4 The slicer

The slice run slices each FDM plate with `curaslice.py` and CuraEngine 5.13.0. `curaengine.sh` downloads the official AppImage of Cura 5.13.0 from GitHub and checks its SHA-256. Then it extracts CuraEngine and its libraries. The cache of GitHub Actions keeps them, so only the first slice run downloads the AppImage (330 MB).

The slice run slices each resin plate with `resinslice.py` and PrusaSlicer 2.8.1. `prusaslicer.sh` downloads the official AppImage of PrusaSlicer (99 MB) and checks its SHA-256, and the cache keeps it too. PrusaSlicer slices the parts of the plate as one model, with its supports and its pad. `resinslice.py` writes the layers as a `.cbddlp` file of version 4 for the ChiTu board of the Phenom, as ChiTuBox 1.4.0 writes it.

The Cura app works out the value of each setting before it calls CuraEngine. `curaslice.py` does the same work with the rules of Cura 5, in the Python standard library:

- The stack of the printer, top first: the settings of the part, the quality changes (the profile), the quality, the material (extruder only), the definition changes, and the definition with its parents.
- The extruder stack falls through to the global stack. A setting that is not settable per extruder comes from the global stack.
- A formula gets the values of its names from the stack that asks. The functions are those of Cura, for example `extruderValue` and `resolveOrValue`.
- The `{key}` tokens of the start and end G-code come from the first extruder.
- The script writes the real header (time, filament, size) from the log of CuraEngine, as the Cura app does.

The folder `profiles/taz-workhorse-se` of Printdeck is the complete Cura stack of the TAZ Workhorse with the SE tool head. `slicer/export_cura_profile.py` copies it from the Cura app of the owner, and stops when a file has a secret. A repository copies the folder of each profile that it uses into its own `profiles/`. So a build depends only on its commit, and a pull request can change a profile. The folder

`profiles/peopoly-phenom` has `resin.json`: the printer (276.48 × 155.52 mm, 3840 × 2160 pixels, a margin of 4 mm) and the resin profile `abs-like-0.05`, with the settings of ChiTuBox for the resin.

A setting of a part that Cura allows for each mesh stays with its part. Another setting of a part applies to the whole plate. A 3MF file keeps the per-mesh settings of Cura, for example a support blocker. A part with a `prebuilt` file, for example a `.cbddlp` file from ChiTuBox, needs no slice. The Action copies that file as the file of the plate.

> **The check against Cura.** `profiles/README.md` compares a slice of the Cura app with a slice of `curaslice.py` of the same STL (PLA 0.10 Fine, roll 4, a brim of 12 lines, tree supports). The print time (12399 s), the filament (2.07922 m), the size and the layers (738) are the same. Of the 211,427 lines of moves, 245 places differ by 0.001 to 0.005 mm.

6.5 The profile version in each G-code

Each profile has a version: the first 12 hex digits of the SHA-256 of the paths and the contents of all the files of its folder. A change to any file of the folder gives a new version. The plan has the version for each part, for example `taz-workhorse-se/pla-0.10-fine@44606b4e58b6`.

The header of each G-code has the line `;Printdeck profile:` with this value. It also has `;Printdeck title:`, `;Printdeck build:` and `;Printdeck commit:`, and the material lines of the Printdeck plugin for Cura. The API keeps the profile from the slice run only when it starts with the name of the profile of the plan, before `@`. The build page shows the profile with its version for each plate.

6.6 The Action: printdeck-action

The Action is the public repository `beetlebugorg/printdeck-action`, under AGPL-3.0. A workflow uses it as `beetlebugorg/printdeck-action@v1`. It is a composite action with six files: `action.yml`, `printdeck_action.py`, `curaslice.py`, `curaengine.sh`, `resinslice.py` and `prusaslicer.sh`. Its inputs are `mode` (plan or slice), `build`, `config` (default `printdeck.yml`) and `api` (default `https://printdeck.beetlebug.org`).

The workflow in its README has the permissions `contents: read` and `id-token: write`, and two jobs:

- **plan** runs on a push of each branch and of each tag `v*`. The workflow has no trigger `pull_request`. The run checks out the full history with no stored token, and compares a branch with its merge base with the default branch, and main with the commit before the push. A part changed when its files, its prebuilt file, the folder of its profile, or its entry in `printdeck.yml` changed. A tag has no base, so its plan marks no part.
- **slice** runs on `workflow_dispatch` with the inputs `build` and `title`. The title names the plates, and GitHub shows it as the name of the run. It gets the plates, and sends its run ID to `builds/<id>/started`. The build page then links to the run. A failure sends `builds/<id>/failed` with the error.

The Action needs no secret. For each request, it asks GitHub for a new OIDC token, with the audience `printdeck.beetlebug.org`: the host name of the input `api`.

The GitHub App. The service starts each slice run with its GitHub App, when the stack has the secret `<stack>-github-app` (`pwf github-app`). For each build, the API signs a JWT of the App and finds the installation of the App on the repository. It gets an installation token for that one repository, with the permissions Actions (write) and Metadata (read). Then it reads the default branch, and starts `printdeck.yml` on it with the inputs `build` and `title`. A workflow from before the input `title` gets the

build only. When GitHub refuses, the build fails with the reason. With no App, the build page shows the owner the command `gh workflow run printdeck.yml -R <repo> -f build=<id>`, with Copy.

6.7 Security of the builds

1. **The token.** Each request of the Action has an OIDC token of GitHub in the header `Authorization`. The API checks its RS256 signature with the keys of GitHub, from `https://token.actions.githubusercontent.com/.well-known/jwks`. It keeps the keys for 1 hour, and reads them again for a key ID that it does not have. It checks the issuer, the audience (the host name of the service) and the expiry, with 1 minute of leeway. The token must have the claim `repository`.
2. **One use.** The API writes `oidc/<SHA-256 of the jti>` with a conditional write that fails when the file exists. A second use of a token gets 401, and so does a token with no `jti`. A lifecycle rule deletes these files after 1 day.
3. **The link to the repository.** The owner links a repository on the project page (`PUT projects/<id>/repo`). The API writes `repos/<owner>/<name>.json` in lower case, with the project. The repository of the token selects the project, and the project must still name that repository. Otherwise the request gets 403. A second project cannot link the same repository (409). A plan or a build of another repository gets 403.
4. **The reviewed workflow.** Each request on `/api/v1/builds/` needs a token with the `job_workflow_ref` `<repo>/.github/workflows/printdeck.yml@refs/heads/main`, or the same on master. A change to that file needs a push or a merge to that branch. A copy of the workflow on another branch or in a pull request can send a plan, and gets 403 on a build.
5. **Forks.** The run of a fork sends a plan with the token of the fork. Section 6.8 gives the checks. A fork never fills a build: its token has another `job_workflow_ref`.
6. **The uploads.** The grant of a plan or a build has one folder, 200 MB for each file, and 1 hour. Done checks that each named file is in the folder. Each file of a build needs a size of at most 200 MB and a SHA-256 in hex.
7. **The people.** Only the owner links a repository, slices a plan, approves or rejects a plate, and sends a plate to a printer. Members see the plans and the builds. The routes that show a plan or a build refuse a deck.
8. **The App.** Each installation token of the App works for one repository, with only the permission of its call: start a workflow, read the pull requests, or write the comment on a pull request. The private key of the App is in Secrets Manager. The role of the function can read that secret only when the stack has the App.

Section 4.2 lists the limits of the builds.

6.8 Pull requests from forks

A member can work in a fork of the repository of the project. The repository of the project runs nothing for a fork. The workflow of the fork runs in the fork, and asks GitHub for an OIDC token. That token names the fork, its owner, the branch and the commit. Printdeck checks the signature. Then it decides by its own rules if it trusts the token.

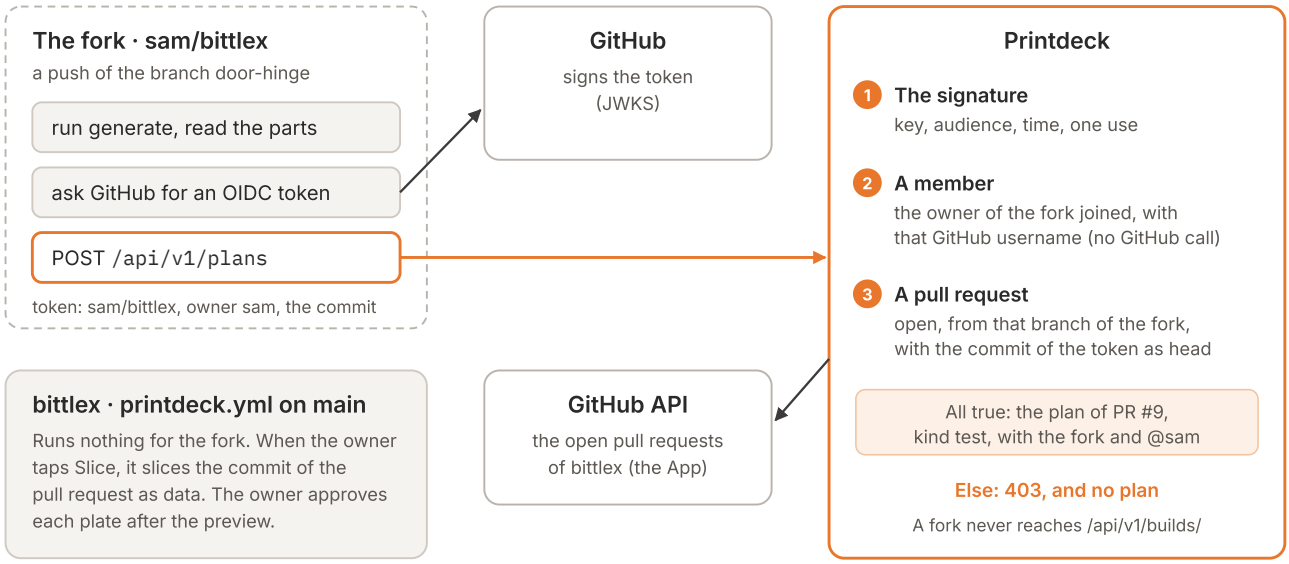


Figure 24. The plan of a member's fork. The fork sends the plan with its own token. Printdeck checks the token, the member and the pull request.

Check	What it proves	Where the facts come from
1. The signature, the audience, the time, one use	GitHub made the token for Printdeck a few minutes ago, and nobody used it before.	The token, and `oidc/`
2. The owner of the fork is a member	The owner invited this person, the person joined, and the owner set this GitHub username.	`repository_owner` of the token, and the members of the projects. A stranger stops here.
3. An open pull request with that commit	The fork is a fork of the repository of the project, the member opened a pull request, and the plan is of its newest commit.	The GitHub API, with an installation token of the App on the repository of the project

The API looks for a member's fork only after a token of an unknown repository. `planDone` then accepts a token of the fork for the plan of the fork only. A push to a fork with no pull request, a pull request of a stranger, or an old commit of the branch makes no plan.

The member forks the repository, turns on the Actions of the fork once, pushes a branch and opens a pull request. Each new push of the branch sends a new plan. The owner sets the GitHub usernames on the card People of the overview (Edit), or in Settings.

6.9 What changed in the models

The page compares each changed model of a plan with the same model in the newest older plan of main or of a tag, else the newest older plan. The Action saves each model in the middle of the bed at Z 0, so the page puts the old model at the middle, or at the low or the high side of each axis, and keeps the place with the least volume that differs (a grid of 48 cells on the longest side). Then it compares the two models in a grid of 140 cells, from the crossings of the surface along Z in each column.

- The plans list has a line for each changed model: the change of the size, and of the volume in cm³ and in %. The models with the same shape (a change of the profile or of the settings only) share one short line.

- The plan page has a card What changed. See in 3D shows the new model faint, the added material in green and the removed material in red.

6.10 The comment on a pull request

The service keeps one comment on each pull request, with its GitHub App. The record `x/<project>/prs/<n>.json` has the ID of the comment, the state of the pull request, and the public copies of the photos. When the comment is gone, the service makes a new one.

- The comment names the commit of the plan, the count of the changed parts, and the plan that it compares with. A table gives what changed in each model, with the comparison of section 6.9 in Go (`cloud/modeldiff`).
- It has a picture of each changed model, at most 8: gray for no change, green for new material, red for material that is gone.
- It shows the newest 3 builds of all the plans of the pull request. Each plate has its parts, its time, its material and its state. An FDM plate has a picture of its G-code. A print of a plate gives its result and its first photo.
- The pictures are public files `gh/<32 hex digits>.png` (and `.jpg` for a photo), so GitHub shows them for a private repository too. The names are random.
- The service writes the comment again when a plan is ready, when a build starts, fails or is ready, at an approval, a rejection or a send, when a deck takes a plate, and when a print of a plate ends or gets a photo.

The API writes the comment after the request, in a job. `Server.Later` invokes the function again with the event `{"pwf":"job"}`, and `RunJob` writes the comment in at most 50 seconds. The job has no retries, and an event waits at most 600 seconds. With no `Later`, or when the invoke fails, the job runs in the request, for at most 20 seconds. A failure goes to the log only.

6.11 What is not built yet

- **Commit status:** the service sets no commit status on a pull request.
- **Print records:** the print page and the print tiles do not show the build or the pull request of a plate.
- **Resin pictures:** a resin plate has no picture in the comment on a pull request.

7 The print queue

Each deck keeps a queue of files. The queue never starts a print: at the end of each print, and when a file comes into the queue while the printer is free, the deck asks. A press on its touch screen starts the next file.

7.1 The queue of a deck

The deck keeps the queue in `queue.json` of its state folder (`internal/queue`), with at most 50 files. A file has its path, its name, its project, its material and color, the print time of the slicer, and where it came from: the deck, a slicer, or a build.

- **At the end of a print.** When the printer goes from a print to no print and the queue has files, the deck asks on each screen: Next, with Start, Skip and Later. Start runs the same material check as Print, and puts the print in the project of the file. Skip takes the file out, and asks for the next one. The paired phones get the push message Next in the queue, with the name, the material, the time, and what to do before Start.
- **The Queue page.** Machine has a Queue page: the files in their order with their material and time, and the total time. A tap on a file starts it, moves it to the front or up, or takes it out.
- **The file page.** Queue adds the file to the end of the queue, also during a print.

7.2 Files from the slicers

- **Cura and the TAZ.** A file that Cura sends with print during a print goes to the queue, and the upload gets 201 with no start. When the queue has files and the printer is free, the deck asks on its screens: print it now, add it to the queue, or start the queue.
- **ChiTuBox and the Phenom.** A press on Print in ChiTuBox during a print adds the file to the queue, and ChiTuBox gets an answer that names the queue. With files in the queue, the deck asks on its screen. No start from ChiTuBox reaches the board then: only a press on the deck sends M6030. A queued resin file has the print time of its header.
- **Reprint on the Phenom** uses the file on the board. The key of the history is not a file: with it, the deck says that it does not know the file.

7.3 Plates of the builds

On the build page, an approved plate (FDM or resin) has Send to for each printer of its process (`POST …/plates/<n>/send`). The service keeps an index entry in `printers/<id>-plates/`. Each minute, the deck of the printer reads its plates (`GET printers/<id>/plates`, presigned addresses of the bucket for one hour, because a deck has no cookies of the service), downloads each file, checks the size and the SHA-256, adds it to its queue with the names of its parts and its project, and says so (`POST printers/<id>/plates/taken`). An FDM deck keeps the file in `builds/` of its G-code folder. A resin deck keeps the file (a `.cbddlp` of ChiTuBox) with its copies of the resin files, and sends it to the board in the background when no print runs. The board gets no start. The build page then says that the plate is in the queue.

7.4 What a project sees

The deck reports its queue at each change, and each 2 minutes while the queue has files (`POST printers/<id>/queue`): the files of projects of the service, each with its place and the estimate of its start after the print now and the files before it. The service keeps the files of each project in the project. A

project sees its own files, their places and their starts, and not the other files. The owner can take a file of the project out of a queue with Remove on the project page (`DELETE projects/<id>/queue/<printer>/<item>`). The service keeps the removal in `printers/<id>-remove/` for at most 7 days. Each minute, the deck reads its removals with its plates (`GET printers/<id>/removes`), takes each file out when the project matches, and says so (`POST printers/<id>/removes/done`). The file stays on the deck. The owner also gets the state of each printer, with a project or not (`GET printers`, `live`): the print, its progress and time left, the last result, and the count and the time of the files of the queue.

8 The apps

The iPhone and Watch app in `watch/` reads the decks on the home network and the service away from home. `docs/specs/iphone-app.md` has its phases.

8.1 The deck API for apps

Both decks serve the OctoPrint-compatible API with the API keys of the deck. `GET /api/plugin/printdeck` has the status, the LCD panel, `process` (`fdm` or `resin`), and `printer_id`: the ID of the printer at the service. The queue routes list the files (`GET …/queue`), move a file (`POST …/queue/<id>/move`) and take a file out (`DELETE …/queue/<id>`). The API has no start of a queued file.

On the resin deck (`newResinAPI`) the API has the status, pause and cancel, the push pairing and the camera. It refuses uploads, each start and each resume, because the UV light goes on only after a press on the touch screen.

8.2 The iPhone and Watch app

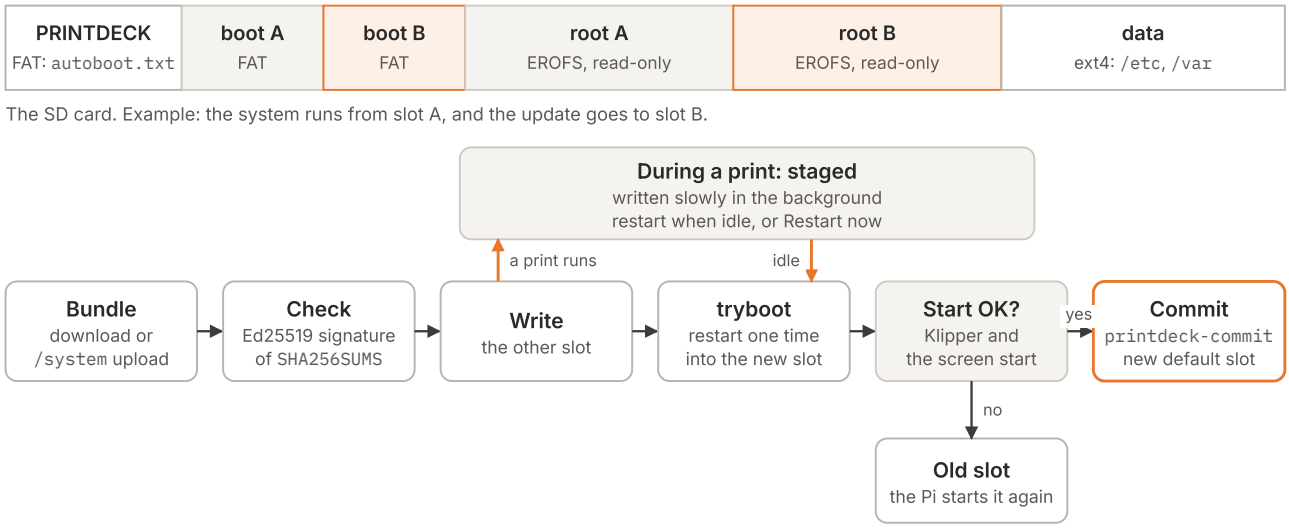
- **Several printers.** The app keeps a list of decks, each with its key in the keychain group of the app. The app has two tabs, Prints and Projects. The Prints tab has a card for each deck. The Live Activity shows the print that ends first. The widgets and the watch show the deck that prints.
- **Away from home.** The app reads the service: the live prints of the projects, and for the owner the state of each printer. It matches them to its decks with `printer_id`.
- **Projects.** The overview of a project, the approval of plates, Send to a printer, the marks Test, Ready, Delivered and Discarded (with a note of why), notes, and photos from the camera. The 3D preview opens the web page in a web view, with the session. The Projects tab has the sign-in, and a card for each project with its prints that run, the prints ready for pickup, the plates that wait and the files in the queues. The app loads the photos and the models with the signed cookies that the replies of the API set.
- **The queue.** The queue of each deck, with Up, Down and Remove, and no Start. A push message Next in the queue opens its deck. The app registers with the push relay at `/api/v1/relay/` of the service, and the decks post their encrypted messages there.
- **The resin deck.** The app has no Print and no Resume for a resin deck.

8.3 The sign-in in the app

The service serves `/.well-known/apple-app-site-association` with the webcredentials of the app. iOS then lets the app use the passkeys of printdeck.beetlebug.org: the passkeys of the iPhone and of a password manager, a passkey of another device with a QR code, and a security key. The app uses the routes of the web page (`auth/login/begin` and `finish`) and keeps the session token in its keychain. A session works for 14 days. A member gets the projects of the member, and no state of the printers.

9 Updates

The disk has two system slots. An update writes the slot that does not run, and the Pi tries it once before it keeps it.



The SD card. Example: the system runs from slot A, and the update goes to slot B.

Figure 25. The A/B update. The firmware of the Pi selects the slot with `autoboot.txt` and `tryboot`.

9.1 The image and the slots

Each slot has a boot partition (FAT) and a root file system (EROFS, read-only). The data partition (ext4) is a writable layer over `/etc`, `/var`, `/home` and `/root`. It keeps the identity, the history and the settings of the deck across updates. The deck looks for a new GitHub release one minute after the start, and then once a day. Machine, Software update on the touch screen installs it. The page `/system` also accepts an update bundle.

9.2 The steps

1. The system helper checks the Ed25519 signature of `SHA256SUMS`, and then the SHA-256 of each file of the bundle.
2. It writes the other slot.
3. With no print, the Pi restarts with `tryboot` into the new slot, one time.
4. When Klipper and the touch screen start, `printdeck-commit` changes `autoboot.txt`, and the new slot becomes the default.
5. When they fail, the next restart starts the old slot again. `panic=10` restarts the Pi after a kernel panic.

9.3 An update during a print

An install during a print does not stop the print. The helper writes the other slot at 4 MB a second, with nice 19 and the idle I/O class, because Klipper reads the G-code from the same SD card. The update is then staged: `/run/printdeck-update/staged.json` names it, and its root file system is mounted read-only at `/run/printdeck-next`. `autoboot.txt` does not change.

The deck restarts into the new slot when the printer is idle after the print. Idle means no print, no heater with a target, no touch for 2 minutes, and the bed at 35 °C or cooler. After 30 minutes, a warm bed does not stop the restart.

> **Restart the screen now.** During the print, this key runs the deck program of the update in place of the old one. First the deck checks that the new program has the same interface version as the running Klipper host. Then it writes the time lapse frames and the state of the share link to RAM, and ends. The new program reads them, so the time lapse and the share link continue. The screen is dark for a few seconds, and Klipper keeps the print. After the print, the Pi restarts into the new slot, and Klipper then updates too.

On resindeck, the ChiTu board prints a file from its own storage, after the deck sent it. The deck reads the progress over UDP.

A API routes

All routes are under `/api/v1/`. The largest body is 64 KB. A request with no valid auth gets 401, and a missing thing gets 410. A reply that links to files of a project also sets the signed cookies of that project. The routes of the Action need a new OIDC token of GitHub for each request.

Route	Auth	What it does
`GET health`	none	Reports that the API runs.
`POST decks`	enrollment code	Enrolls a deck with its public key.
`GET`, `DELETE decks/self`	deck signature	Shows the deck, or removes it (Disconnect).
`POST owner/code`	deck signature	Makes an owner code (10 minutes).
`POST auth/owner/begin`, `finish`	owner code	Makes an owner passkey.
`POST invites/view`	invite code	Shows the project and the owner of an invite.
`POST auth/invite/begin`, `finish`	invite code	Makes a member passkey, and joins.
`POST auth/login/begin`, `finish`	passkey; `finish` accepts an optional invite code	Signs in, and gives a session token. With an invite code, a member passkey also joins the project of the invite.
`GET me`	session	Shows the person of the session.
`GET`, `POST printers`	owner or deck	Lists the printers. A deck registers its printer.
`GET names/<slug>`	session	Finds the project of a slug. With `?view=1`, it also gives the view of the project, so a project page needs one request.
`GET`, `POST projects`	session or deck	Lists the projects, or makes one (owner). A member gets each project of the session that still has the member.
`projects/<id>` and below	session or deck	View, rename, delete; invites, members, public address; prints, live tiles, photos. The rows below are the routes of the notes, the prints, the photos, the queue and the builds.
`PATCH projects/<id>/notes`	session	Changes the notes of the project.

Route	Auth	What it does
`PATCH projects/<id>/prints/<print>`	session; owner for title, marks, source	Changes the title, the notes, the marks Test, Ready, Delivered and Discarded (with a note of why), or the source of a print. A member changes the notes only.
`PUT projects/<id>/photos/<photo>`	session or deck; a member moves only their own photos	Moves a photo to another print of the project, or to no print.
`PATCH projects/<id>/members/<member>`	owner or deck	Sets the GitHub username of a member who joined.
`PUT projects/<id>/repo`	owner or deck	Links a GitHub repository, or removes the link with "".
`GET projects/<id>/plans/<plan>`	session	Shows a plan with the paths of its models, and its builds.
`GET projects/<id>/plans`	session	Lists the plans of pull requests and of tags.
`POST projects/<id>/plans/<plan>/builds`	owner or deck	Makes a build with the plates, and starts the slice run with the GitHub App.
`GET projects/<id>/builds/<build>`	session	Shows a build with its plates, and the `gh` command when the stack has no App.
`POST projects/<id>/builds/<build>/plates/<n>/approve`, `reject`	owner or deck	Approves a plate, or rejects it with a reason.
`POST plans`	OIDC token of the linked repository, or of a member's fork with an open pull request	Makes a plan, and gives the grant of its models and its PDFs.
`POST plans/<id>/done`	OIDC token of the repository of the plan, or of its fork	Checks the models and the PDFs, and marks the plan ready. An older waiting plate with a changed part becomes "replaced". With the GitHub App, the service then writes its comment on the pull request.
`GET builds/<id>`	OIDC token of `printdeck.yml` on main or master	Gives the commit and the plates of the build.
`POST builds/<id>/started`	the same	Records the run ID. A failed build goes back to "slicing".
`POST builds/<id>/progress`	the same	Records a line of progress of the slice run. The build must be "slicing".

Route	Auth	What it does
`POST builds/<id>/files`	the same	Records the files of the plates, and gives the grant of the build.
`POST builds/<id>/done`	the same	Checks the files. The build becomes "ready", and each plate "waiting".
`POST builds/<id>/failed`	the same	Marks the build "failed", with the error.
`POST shares`	deck signature	Makes a share, with its first grant.
`shares/<id>/grant`, `end`, `project`, `DELETE`	signature of the deck of the share	A new grant, the end, the project of the share, or a delete.
`shares/<id>/ready`, `photo`	owner key, or the deck	The Ready flag and note; photos.
`POST projects/<id>/builds/<build>/plates/<n>/send`	owner or deck	Sends an approved plate to a printer of its process.
`GET printers/<id>/plates`, `POST printers/<id>/plates/taken`	signature of the deck of the printer	The plates that wait for the printer, with presigned addresses of the bucket; the plate is in the queue.
`POST printers/<id>/queue`	signature of the deck of the printer	The files of the queue that are in projects: their places and starts.
`DELETE projects/<id>/queue/<printer>/<item>`	owner or deck	Asks the deck to take a file of the project out of its queue. The file must be in the last report of the queue.
`GET printers/<id>/removes`, `POST printers/<id>/removes/done`	signature of the deck of the printer	The files that the owner took out on the project page; the deck took them out.
`POST printers/<id>/live`	signature of the deck of the printer	What the printer does now. `GET printers` has it for the owner.
`/.well-known/apple-app-site-association`	none (a file of the bucket)	The webcredentials of the iPhone app, for the passkey sign-in in the app.
`relay/register`, `relay/push`	none for register (a code comes by push); the send token for push	The push relay of the iPhone app, when the stack has the secret of the relay. The app registers here, and the decks post their encrypted messages here.