

Test Plan: Team API Keys & Proxy Pools

QA approach for the new API keys feature

1. Test Plan

Core Features that Must Pass

Feature	What it does	What to check
Create a key	User types a label, selects a product (Residential, Premium, Dedicated), and picks an environment (dev/stage/prod)	API returns 201, key is masked (only shows last 4 chars), key is stored in the DB, UI shows it instantly
View a key	Shows label, assigned products, environment, and rate-limit numbers	Full secret is never shown; UI displays label & masked key
Edit a key	Change label, add/remove product assignments, tweak rate-limit caps	PATCH works, DB updates, UI reflects new values right away
Revoke a key	Stops the key from being usable	DELETE returns 204, subsequent calls to the proxy return 401/403 within 5 s
Bulk actions	Create/revoke 10k+ keys in one request	Operation finishes in < 90s, no race conditions, UI updates for all affected keys
Logs	Every request made with a key is recorded (time, status, IP, country)	Logs show correct counts, last request details; filtering by key works; no performance hit even with 150k keys
Environment rules	dev → low limits, no IP whitelist. stage/prod → higher limits, IP whitelist enforced	A dev key used on prod should be rejected; IP whitelist is enforced for prod keys
Proxy routing	Requests are routed to the pool that the key was assigned to	If a key has multiple assignments, the header overrides the default pool

Core Test Cases

UI:

- Create key with label/assignment; verify instant list update, pagination/search/filtering works with 100+ keys
- Bulk revoke 1k keys; confirm UI reflects changes, no UI jank on large datasets
- View logs: Filter by key, verify real-time updates after simulated requests

API:

- `POST /api-keys` : Returns 201 with masked key; assign to products/envs
- `GET /api-keys/{id}` : Returns details (no full key); pagination handles 150k with correct counts
- `DELETE /api-keys/{id}` : 204 response, subsequent gateway calls fail
- `PATCH /api-keys/{id}` : Update label/assignment/rate limits; validate changes propagate

Data:

- DB integrity: Query backend for key count/assignments post-CRUD; ensure no orphans after revocation
- Logs accuracy: Simulate 1k requests via key, verify counters/IP/country match; reset on billing cycle
- Scale: Seed 150k keys, test query performance (P95 <500ms), concurrent creates (no races)

Exploratory Angles (Proxies/Rate Limits/Abuse)

- **Proxies:** Test pool routing edge cases (e.g., key assigned to multiple pools—header overrides default); geo-leakage (wrong country IP); sticky sessions for Dedicated
- **Rate Limits:** Burst traffic exceeding limits (e.g., 1001 req/s on prod key → 429 with Retry-After); cross-env abuse (use dev key in prod → rejection)
- **Abuse:** Brute-force key guessing (rate-limit API); concurrent revokes during active use; IP whitelist bypass attempts (spoofed IPs); high-volume logs (1M+ entries/key) for overflow/injection
- **Usability:** Multi-user concurrent edits (optimistic locking); error messaging clarity on failures; accessibility (screen reader on masked keys)

System Flow (how I understand it):

User Creates API Key → Assigns to Proxy Pool(s) → Key Used in Request → Gateway Routes to Pool → Proxy Request Executed → Usage Logged

Environment rules & rate limits enforced at gateway | IP whitelist checked before routing

2. Automation Prioritization (1 day + 2 QAs)

Given the tight timeline, I'd focus on automating the critical paths first—stuff that covers the main user flows and high-risk areas like revocation and scale. With me + 1 senior QA + 1 junior QA, we can probably knock out ~50% automation coverage (5-6 cases) on day 1.

Framework choice: Cypress for UI (I've used it a lot and it's reliable), Postman/Newman for API, and SQLAlchemy for data checks. I'd own the complex stuff—scale tests, concurrency checks, and code reviews. Senior QA can handle API integrations, and I'll get the junior QA started on simpler UI scripts with some guidance.

What to automate first & why

Test	Auto?	Reason	Who
UI: Create single key	Yes	High-frequency action, quick to script, catches regressions early	Junior (I review)
UI: Bulk create (10k+ keys)	Yes	High-scale risk, critical for enterprise users	Me (complex)
UI: Bulk revoke	No	Manual first to validate UX, automate later if needed	Manual
UI: View/filter logs	No	Good for exploratory first, automate in future sprints	Manual
API: POST /api-keys	Yes	Core creation, easy to parameterize, high ROI	Senior QA
API: GET /api-keys	Yes	Scale-critical (150k keys), verifies pagination	Me (scale setup)
API: PATCH /api-keys/{id}	Yes	Ensures updates propagate, covers security controls	Senior QA
API: DELETE /api-keys/{id}	Yes	Critical security—revocation must be immediate	Me (high risk)
Data: DB integrity	No	Better as manual queries first, automate for CI later	Manual
Data: Logs accuracy	No	Time-sensitive (billing), high effort for low initial gain	Manual
Data: Concurrent operations	Partial	Hard to automate reliably, use Locust for load testing	Exploratory

Summary: First 6 automations are UI1, UI2, API1-4. Focus on API-heavy flows (faster, less flaky) covering ~80% of risks. Morning: planning/setup, parallel scripting during the day, afternoon: test runs & reviews. This leaves time for manual exploratory testing on proxies/abuse scenarios. If we run short on time, we'll drop UI2.

3. CI Integration (Cypress + GitHub Actions)

I've wired up Cypress with GitHub Actions before and it's pretty straightforward. Here's how I'd set it up:

Folder structure

```
project-root/ ├── cypress/ | ├── e2e/ // test files like createKey.cy.js | ├── fixtures/
// mock data | ├── support/ // custom commands (login, DB seeding) | └──
cypress.config.js ├── .github/workflows/ | └── ci-tests.yml ├── package.json └──
README.md
```

Running locally is just `npx cypress run`

CI Jobs (pseudo-YAML)

Job 1: Smoke tests

```
name: CI Tests on: [push, pull_request] jobs: smoke: runs-on: ubuntu-latest steps: -
checkout code - setup node v20 - npm install - npx cypress run --spec
"cypress/e2e/smoke/*" --env baseUrl=$TEST_URL - upload videos/screenshots on failure
```

Job 2: Full E2E (runs only if smoke passes)

```
full-e2e: needs: smoke runs-on: ubuntu-latest steps: - checkout code - setup node - npm
install - seed DB with 150k keys - npx cypress run --browser chrome --parallel --record -
-key $CYPRESS_KEY - generate & upload Mochawesome reports
```

4. QA Metrics

1. Core Flow Pass Rate (create → assign → use → revoke)

Target: $\geq 95\%$. If it drops below 90%, we don't deploy. Main user journey needs to be solid.

2. Flaky Test Rate

Target: $< 5\%$. Flaky tests kill trust fast, so anything above 5% gets quarantined and fixed or deleted immediately.

3. Dashboard Regression Rate (new failures after changes)

Target: $< 2\%$. A spike means something big broke (like pagination with 100k keys) and we need to fix it before merging.

5. Environment Handling

Test Data Strategy

- **Dev:** Fake scripted data—quick-generated keys, small proxy pools (10-100 IPs). Automation hooks spin it up fast.
- **Stage:** Prod-like data (50k keys from snapshots or generators). Cycle proxies regularly to catch scale issues.
- **Prod:** Hands-off—monitoring, canary releases, low-risk flagged checks only.

Config & Secrets

- Keep configs (rate limits, whitelists) in AWS Secrets Manager—inject during builds, never hardcode
- Rotate secrets (DB creds, keys) every 90 days, monitor access logs

Key Risks per Environment

- **Dev:** Too-loose controls → test data leaks. Fix: mocks + role-based access
- **Stage:** Heavy loads → crashes. Fix: add limits & alerts
- **Prod:** Outages, key leaks. Fix: gradual rollouts, monitoring (Datadog), easy rollbacks