ONE NPUB — Demo Steps Guide (UI + CLI)

ONE NPUB — Demo Steps Guide (UI + CLI)

Version 2.1 Stable | March 2026 | Thomas+Agent21

Each step shows: **Use Case** → **What Happens** → **Crypto/Algorithm** → **UI Button** → **CLI Command** → **Expected Result**

Step 0 — 👋 Welcome

Use Case: Introduction to ONE NPUB — threshold signing for Nostr.

What Happens: Welcome screen. No execution. Shows all participants: Coordinator, Agents A–H, Policy Engine, strfry Relay.

Crypto: FROST (RFC 9591) — Flexible Round-Optimized Schnorr Threshold Signatures over secp256k1, producing BIP-340 compatible output. NIPs used: NIP-01, NIP-03, NIP-19, NIP-85.

UI: Click “Next →”

CLI:

```bash
bash cli-demo/step-00-welcome.sh
# Or manually:
curl -s http://localhost:3333/status | jq .
```

Expected Result:

```
✅ Coordinator reachable (port 3333)
✅ Docker Proxy reachable (port 3334)
✅ strfry Relay reachable (port 7777)
✅ All systems ready.
```

Step 1 — Genesis: 1/1 FROST Key

Use Case: Coordinator creates a solo key — equivalent to a classic Nostr `nsec`, but inside the FROST framework. Starting point for all threshold operations.

What Happens: DKG (Distributed Key Generation) runs with a single participant. The coordinator holds the full key. A `npub` is generated that will remain identical across all future reshares.

Crypto: - Polynomial of degree 0: `f(x) = s` (constant = the secret) - Group public key: `P = s · G` (secp256k1 generator point multiplication) - NIP-01 event format: pubkey = 32-byte x-only public key

UI: Click “Execute →”

CLI:

```bash
bash cli-demo/step-01-genesis.sh
# Or manually:
curl -s -X POST http://localhost:3333/genesis \
  -H 'Content-Type: application/json' \
  -d '{"tier": 2}' | jq .
```

Expected Result:

```
✅ Genesis complete — Epoch 1
npub:
ecdfda6eb1d94649f00a13a60c39317a9a9abb19fdcc9a7df4dc4061116846a8
Coordinator holds full key — single point of failure (for now)
```

Step 2 — Deploy Agents A–D & Reshare to 5/7

Use Case: Split the key across 4 agents. No single device holds the full secret anymore. The coordinator needs 2+ agents to sign.

What Happens: Four Docker containers start sequentially. Each agent registers with the coordinator via the strfry relay. A fleet-reshare redistributes the key: coordinator gets 3 shares (indices 1, 100, 101), each agent gets 1 share. Threshold becomes 5/7.

Crypto: - Threshold model: `t/n = (a+1)/(2a-1)` → with 4 agents: `t=5`, `n=7` - Coordinator shares: `a+1-k = 4+1-2 = 3` - Reshare: new degree-4 polynomial over same `f(0)`, same group public key - Shamir's Secret Sharing: `f(i)` evaluated for each participant index `i` - Lagrange interpolation: any 5 of 7 points reconstruct `f(0)`

UI: Click "Execute →"

CLI:

```bash
bash cli-demo/step-02-deploy.sh
# Or manually:
curl -s -X POST http://localhost:3333/docker/start-agent \
  -H 'Content-Type: application/json' -d '{"agentId":"a"}'
# (repeat for b, c, d — wait 15s each)
curl -s -X POST http://localhost:3333/fleet-reshare \
  -H 'Content-Type: application/json' \
  -d '{"agentIds": ["a","b","c","d"], "k": 2}' | jq .
```

Expected Result:

```
✅ Agent A online
✅ Agent B online
✅ Agent C online
✅ Agent D online
✅ Reshare complete — Epoch 2
Model: t=5, n=7, coord=3, agents=4
npub: ecdfda6eb1d94649...  (same as genesis!)
```

Step 3 — Sign Kind 1 (Text Note) ✅ autonomous

Use Case: Normal text note publishing. Policy allows fully autonomous threshold signing — no human approval needed.

What Happens: Coordinator selects a quorum (its 3 local shares + 2 online agents), runs FROST 2-round signing, aggregates partial signatures into a single BIP-340 Schnorr signature. Event is published to strfry relay.

Crypto: - FROST Round 1: each participant generates nonce pair `(d, e)`, publishes commitments `(D = d·G, E = e·G)` - FROST Round 2: each participant computes partial signature `z_i = d_i + e_i·ρ_i + λ_i·s_i·c` - `s_i` = secret share, `c` = BIP-340 challenge hash, `λ_i` = Lagrange coefficient, `ρ_i` = binding factor - Aggregation: `z = Σz_i`, `R = Σ(D_i + ρ_i·E_i)` → `(R, z)` = valid Schnorr signature - NIP-01: `event_id = SHA-256([0, pubkey, created_at, kind, tags, content])`

UI: Click "Execute →"

CLI:

```bash
bash cli-demo/step-03-sign-kind1.sh
# Or manually:
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' \
  -d '{"message": "Hello from threshold key", "kind": 1}' | jq .
```

Expected Result:

```
✅ Kind 1 — verified=true, policy=autonomous
Event:
f7fb9a843ee7b225e334ca918e33e6dc81d50a0afc9c9cb8ba602e2a9c2b416f
Sig:   3e4d606dfd24e495...
```

Step 4 — Sign Kind 7 (Reaction 🤙) ✅ autonomous

Use Case: Reaction event — lightweight social signal. Signs immediately with no approval queue.

What Happens: Same FROST signing flow. Fresh nonces generated — critical for Schnorr security.

Crypto: - Fresh nonce pair `(d, e)` generated for EVERY signature - Nonce reuse attack: if same `d` used twice with different messages, secret key is extractable: $s = (z_1 - z_2) / (c_1 - c_2)$ - This is why FROST mandates fresh commitments per signing session

UI: Click "Execute →"

CLI:

```bash
bash cli-demo/step-04-sign-kind7.sh
# Or manually:
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' \
  -d '{"message": "🤙", "kind": 7}' | jq .
```

Expected Result:

```
✅ Kind 7 — verified=true, policy=autonomous
```

Step 5 — Sign Kind 0 (Profile) ⚠ requires_cosign

Use Case: Profile update — sensitive operation. Policy marks as `requires_cosign`. In production: queued for human approval via push notification.

What Happens: Same FROST signing path, but policy gate logs the elevated trust requirement. In demo mode, proceeds with warning.

Crypto: - Policy enforcement is pre-crypto: evaluated before nonce generation - Policy tiers: `autonomous` (sign immediately) → `requires_cosign` (needs approval) → `forbidden` (reject) - NIP-01 Kind 0: profile metadata event `{"name":"...","about":"..."}`

UI: Click "Execute →"

CLI:

```
bash cli-demo/step-05-sign-kind0.sh
# Or manually:
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' \
  -d '{"message": "{\"name\":\"ONE NPUB\"}", "kind": 0}' | jq .
```

Expected Result:

```
✅ Kind 0 — verified=true, policy=requires_cosign
```

Step 6 — Sign Kind 4 (DM) ✕ forbidden

Use Case: DM signing is blocked by policy. The request never reaches FROST signing — rejected at the policy layer.

What Happens: Coordinator evaluates policy for Kind 4 → `forbidden`. Returns HTTP 403. No nonce generation, no partial signatures, no relay traffic.

Crypto: - Policy preempts all cryptographic operations - Zero FROST rounds executed — instant reject - NIP-01 Kind 4: encrypted direct message (deprecated in favor of NIP-44)

UI: Click "Execute →" — error shown in red

CLI:

```
bash cli-demo/step-06-sign-kind4.sh
# Or manually:
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' \
  -d '{"message": "should be rejected", "kind": 4}' | jq .
```

Expected Result:

```
✅ Kind 4 blocked as expected: Kind 4 is forbidden
```

Step 7 — Agent D Goes Offline

Use Case: Simulate device failure (battery dead, network loss). System must still sign with remaining quorum.

What Happens: Agent D's Docker container is stopped. Coordinator marks D as offline. Then signs with remaining quorum: 3 coordinator shares + 2 online agents (A, B or C) = 5 ≥ threshold.

Crypto: - t-of-n threshold: any valid subset of size ≥ t can produce a valid signature - Lagrange coefficients recomputed for the actual participant set (excluding D) - 5/7 threshold with D offline: coordinator(3) + 2 agents = 5 ✅

UI: Click "Execute →"

CLI:

```
bash cli-demo/step-07-agent-offline.sh
```

```
# Or manually:
curl -s -X POST http://localhost:3333/docker/stop-agent \
  -H 'Content-Type: application/json' -d '{"agentId":"d"}'
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' \
  -d '{"message": "Agent D offline", "kind": 1}' | jq .
```

Expected Result:

```
✅ Agent D stopped
✅ Kind 1 — verified=true, policy=autonomous
✅ Threshold resilience confirmed
```

Step 8 — Sign Kind 9735 (Zap Receipt) ✅

Use Case: Zap receipt — financial event. Threshold signature is indistinguishable from single-signer to any Nostr client or relay.

What Happens: Signs with Agent D still offline. BIP-340 Schnorr output. Published to strfry relay.

Crypto: - Output: 32-byte x-only pubkey + 64-byte Schnorr signature - Nostr clients call `schnorr.verify(sig, event_id, pubkey)` — standard BIP-340 - Zero threshold awareness needed by verifiers - NIP-01 Kind 9735: zap receipt event

UI: Click "Execute →"

CLI:

```
bash cli-demo/step-08-sign-kind9735.sh
# Or manually:
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' \
  -d '{"message": "zap receipt", "kind": 9735}' | jq .
```

Expected Result:

```
✅ Kind 9735 — verified=true, policy=requires_cosign
```

Step 9 — Kind 4 Still Blocked & Agent D Returns

Use Case: Policy is deterministic on event kind — independent of which agents are online. Then Agent D comes back.

What Happens: Kind 4 attempted again → still forbidden (same policy, doesn't care about quorum composition). Then Agent D container is restarted and re-registers.

Crypto: - Policy function: `f(kind) → autonomous | requires_cosign | forbidden` - No dependency on participant set — policy is evaluated before FROST rounds - Agent D re-registration: heartbeat message via strfry relay, coordinator marks online

UI: Click "Execute →"

CLI:

```
bash cli-demo/step-09-kind4-agent-back.sh
# Or manually:
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' -d
'{"message":"test","kind":4}'
```

```
curl -s -X POST http://localhost:3333/docker/start-agent \
  -H 'Content-Type: application/json' -d '{"agentId":"d"}'
```

Expected Result:

```
✅ Kind 4 still blocked (as expected)
✅ Agent D online
✅ Agent D back online — full fleet restored
```

Step 10 — Stress Test (10 Rapid Signatures)

Use Case: Validate throughput under burst traffic. Real-world test of operational capacity.

What Happens: 10 Kind 1 events signed sequentially. Each requires 2 relay round-trips (commitments + partials). Results show timing per signature.

Crypto: - Bottleneck: relay WebSocket round-trip latency (~500ms per round × 2 rounds) - secp256k1 point multiplication: ~1ms (negligible) - Each signature uses fresh nonces — no batching of commitments - All 10 signatures verified with `schnorr.verify()`

UI: Click "Execute →"

CLI:

```
bash cli-demo/step-10-stress.sh
# Or manually:
for i in $(seq 1 10); do
  curl -s -X POST http://localhost:3333/action/sign \
    -H 'Content-Type: application/json' \
    -d "{\"message\":\"stress-$i\",\"kind\":1}" | jq
'{verified,nostrEventId}'
done
```

Expected Result:

```
✅ #1  — 1253ms
✅ #2  — 1280ms
...
✅ #10 — 1362ms
Result: 10/10 passed | Total: ~12900ms | Avg: ~1290ms
```

Step 11 — Add Agent E (6/9 Model)

Use Case: Scale up the fleet. Add a 5th agent and rebalance threshold.

What Happens: Agent E container starts. Fleet-reshare redistributes to 5 agents. Threshold model changes to 6/9: coordinator holds 4 shares, each agent holds 1.

Crypto: - Model: `t/n = (5+1)/(2·5-1) = 6/9` - Coordinator shares: `5+1-2 = 4` - Reshare creates new polynomial of degree 5, same `f(0)`, same npub - All previous shares (from 5/7 polynomial) become useless - Lagrange interpolation adjusts for new index set

UI: Click "Execute →"

CLI:

```
bash cli-demo/step-11-add-agent-e.sh
```

```
# Or manually:
curl -s -X POST http://localhost:3333/docker/start-agent \
  -H 'Content-Type: application/json' -d '{"agentId":"e"}'
# Wait 15s
curl -s -X POST http://localhost:3333/fleet-reshare \
  -H 'Content-Type: application/json' \
  -d '{"agentIds":["a","b","c","d","e"],"k":2}' | jq .
```

Expected Result:

```
✅ Agent E online
✅ Reshare complete — Epoch 3, Model t=6, n=9
npub: ecdfda6eb1d94649...  (unchanged!)
```

Step 12 — Agent E Participates in Signing

Use Case: Verify the new agent received valid shares and can contribute to threshold signing.

What Happens: Kind 1 event signed with Agent E in the quorum. Signature verified under the same npub.

Crypto: - Agent E holds `f_new(6)` — evaluation of new polynomial at index 6 - Lagrange coefficients automatically adjust for any valid quorum including E - Signature verifies under unchanged group public key `P`

UI: Click "Execute →"

CLI:

```
bash cli-demo/step-12-agent-e-signs.sh
# Or manually:
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' \
  -d '{"message":"Agent E check","kind":1}' | jq .
```

Expected Result:

```
✅ Kind 1 — verified=true, policy=autonomous
✅ Agent E successfully integrated
```

Step 13 — Agent B Compromised & Evicted

Use Case: Agent B's device is stolen/hacked. Mark it compromised, stop it, reshare without it. Old shares become cryptographically useless.

What Happens: Agent B marked as compromised (☠). Container stopped. Fleet-reshare runs with remaining agents (A, C, D, E). New polynomial — B's old share can't sign future epochs. Threshold returns to 5/7.

Crypto: - Proactive security: old shares lie on polynomial `f_old(x)`, new shares on `f_new(x)` - `f_old(3)` (B's stolen share) cannot be used with `f_new` shares — different polynomials - Same `f(0)` = same npub, but all coefficients `a₁...a_{t-1}` are fresh random - Threshold: `(4+1)/(2·4-1) = 5/7` (back to 4 agents)

UI: Click "Execute →"

CLI:

```bash
bash cli-demo/step-13-evict-agent-b.sh
# Or manually:
curl -s -X POST http://localhost:3333/signer/3/compromised
curl -s -X POST http://localhost:3333/docker/stop-agent \
  -H 'Content-Type: application/json' -d '{"agentId":"b"}'
curl -s -X POST http://localhost:3333/fleet-reshare \
  -H 'Content-Type: application/json' \
  -d '{"agentIds":["a","c","d","e"],"k":2}' | jq .
```

Expected Result:

```
✅ Agent B marked as ☠ compromised
✅ Agent B stopped
✅ Eviction reshare complete — Epoch 4, Model t=5, n=7
npub: ecdfda6eb1d94649...  (unchanged!)
Agent B's old share f_old(3) is now cryptographically worthless
```

Step 14 — Proactive Key Rotation (Same Fleet)

Use Case: Rotate shares without compromise — password rotation hygiene. Even if someone copied a share yesterday, it's useless today.

What Happens: Reshare among same agents (A, C, D, E). New polynomial, same `f(0)`, same npub. Epoch increments.

Crypto: - `f_old(x) = s + a₁x + ... + a₄x⁴` → `f_new(x) = s + b₁x + ... + b₄x⁴` - Coefficients `aᵢ` and `bᵢ` are completely unrelated (fresh random) - Knowing `f_old(i)` tells you nothing about `f_new(i)` - Same constant term `s = f(0)` → same public key `P = s·G`

UI: Click "Execute →"

CLI:

```bash
bash cli-demo/step-14-rotate.sh
# Or manually:
curl -s -X POST http://localhost:3333/fleet-reshare \
  -H 'Content-Type: application/json' \
  -d '{"agentIds":["a","c","d","e"],"k":2}' | jq .
```

Expected Result:

```
✅ Proactive rotation complete — Epoch 5
npub: ecdfda6eb1d94649...  (still the same!)
```

Steps 15–18 — Post-Rotation Signing (Kind 1, 7, 1, 9735)

Use Case: Business-as-usual signing after compromise response + proactive rotation. Proves rotated shares preserve identity and signing capacity.

What Happens: Four sequential signs — different kinds, different agent subsets participate. All verified=true under same npub.

Crypto: - Threshold quorum flexibility: any valid subset works regardless of composition - Epoch transitions don't affect signature validity — `f(0)` is preserved - Each signature uses fresh nonces from fresh commitments

UI: Click "Execute →" four times

CLI:

```
bash cli-demo/step-15-sign-post-rotate1.sh   # Kind 1
bash cli-demo/step-16-sign-post-rotate2.sh   # Kind 7
bash cli-demo/step-17-sign-post-rotate3.sh   # Kind 1
bash cli-demo/step-18-sign-post-rotate4.sh   # Kind 9735
# Or manually:
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' \
  -d '{"message":"post-rotation #1","kind":1}' | jq .
```

Expected Result (each):

```
✅ Kind X — verified=true
```

Step 19 — 🚨 Emergency Lockdown (ALL Agents Compromised)

Use Case: Worst case scenario. Assume every external agent is compromised. Revoke everything. Generate backup.

What Happens: All agent shares revoked. Key reverts to 1/1 coordinator custody. BIP-39 mnemonic backup generated and displayed once. All agent containers stopped.

Crypto: - Emergency path: new 1/1 DKG, all old polynomial shares invalidated - BIP-39 mnemonic: secret encoded as 12/24 words from 2048-word list with checksum - Recovery: `mnemonic → decode → secret s → s·G = P (npub)` - Deterministic: same secret always produces same public key

UI: Click "Execute →" — mnemonic shown in result

CLI:

```
bash cli-demo/step-19-emergency.sh
# Or manually:
curl -s -X POST http://localhost:3333/emergency-lockdown | jq .
```

Expected Result:

```
✅ 🚨 LOCKDOWN COMPLETE — Epoch 6, Model t=1, n=1
Mnemonic backup: ivy falcon falcon thunder keystone starlight...
All agent shares are now cryptographically dead.
```

Step 20 — Recovery: 1/1 Signing

Use Case: After lockdown, coordinator signs alone to prove immediate operational continuity.

What Happens: Kind 1 event signed in 1/1 mode (coordinator only, no agents needed). Note: in demo, emergency lockdown creates a new DKG (new npub). In production, BIP-85 mnemonic import would preserve the same npub.

Crypto: - 1/1 mode: single share = full secret, `t = n = 1` - Signing is standard Schnorr (no threshold coordination needed) - Identity bound to same key material; custody model changes, not crypto identity

UI: Click "Execute →"

CLI:

```
bash cli-demo/step-20-recovery.sh
```

```bash
# Or manually:
curl -s -X POST http://localhost:3333/action/sign \
  -H 'Content-Type: application/json' \
  -d '{"message":"recovery proof","kind":1,"signerIds":[1]}' | jq .
```

Expected Result:

```
✅ Signed successfully in 1/1 recovery mode
Current npub:  3204a571cc9c9731...
Genesis npub:  ecdfda6eb1d94649...
ℹ Different npub (demo creates new DKG during lockdown)
ℹ In production: BIP-85 mnemonic import → same npub preserved
```

Step 21 — New Fleet (Back to 5/7)

Use Case: Rebuild distributed trust. Start fresh agents and reshare back to original fleet shape. Full operational loop: Genesis → Scale → Compromise → Lockdown → Recovery → Rebuild.

What Happens: Agents A–D started fresh. Fleet-reshare to 5/7. Distributed custody restored.

Crypto: - Fresh shares over current identity secret restore distributed trust - New polynomial, same `f(0)`, new share evaluations for each agent - Complete lifecycle demonstrated: 7 epochs of threshold changes

UI: Click "Execute →"

CLI:

```bash
bash cli-demo/step-21-new-fleet.sh
# Or manually:
for l in a b c d; do
  curl -s -X POST http://localhost:3333/docker/start-agent \
    -H 'Content-Type: application/json' -d "{\"agentId\":\"$l\"}"
  sleep 15
done
curl -s -X POST http://localhost:3333/fleet-reshare \
  -H 'Content-Type: application/json' \
  -d '{"agentIds":["a","b","c","d"],"k":2}' | jq .
```

Expected Result:

```
✅ Agent A online
✅ Agent B online
✅ Agent C online
✅ Agent D online
✅ New fleet online — Epoch 7, Model t=5, n=7
```

Step 22 — Final Verification

Use Case: Audit the full timeline. Verify epoch chain, relay events, and npub consistency.

What Happens: Epoch chain shows all 7 threshold changes. Relay events counted. npub consistency checked across all epochs.

Crypto: - Epoch chain: cryptographic history of all DKG/reshare operations - Each epoch records: threshold, total shares, participant indices - NIP-01 events on strfry relay: independently verifiable by any Nostr client - NIP-03 OTS tags: Bitcoin-anchored timestamp proofs

UI: Click "🏁 Restart"

CLI:

```bash
bash cli-demo/step-22-verify.sh
# Or manually:
curl -s http://localhost:3333/epoch/chain | jq .
curl -s http://localhost:3333/relay/nostr-events | jq .
curl -s http://localhost:3333/status | jq .
```

Expected Result:

```
━━━ Epoch Chain ━━━
Epoch 1: threshold=1, signers=1
Epoch 2: threshold=5, signers=7
Epoch 3: threshold=6, signers=9
Epoch 4: threshold=5, signers=7
Epoch 5: threshold=5, signers=7
Epoch 6: threshold=1, signers=1
Epoch 7: threshold=5, signers=7
Total epochs: 7

━━━ Relay Events ━━━
Published NIP-01 events: 20+

🏁 Demo Complete!
All 23 steps executed successfully.
One npub. Many devices. No single point of failure.
```