OPENPROOF

Deployment & Developer Handbook

From a clean server to a working OAuth/OIDC integration

PART I

Deploy & configure

Installer - PostgreSQL - SMTP - providers - TLS - health

PART II

Build with OpenProof

OAuth/OIDC - APIs - Node.js - PHP - C++ - services

OpenProof 1.1.0-rc1 - 22 September 2026 - Genyleap

docs.genyleap.com/openproof/

How to use this handbook

Operators can complete Part I independently. Product teams can start at Part II once a healthy OpenProof issuer exists. Part III defines the public LLM/MCP documentation boundary. Part IV is a compact troubleshooting index.

Part	Chapter	Sections	What it covers
PART I	Deploy and configure	1-14	Host preparation, installer, database, secrets, delivery, providers, gateway, TLS and health.
PART II	Build with OpenProof	15-40	OAuth/OIDC, PKCE, applications/clients, language integrations, APIs, accounts and service clients.
PART III	LLM and MCP integration	41-43	Machine-readable docs, read-only docs MCP and agent safety.
PART IV	Troubleshooting	44-49	Health, providers, callbacks, 401/403, SMTP and upstream failures.

> Online contract: https://docs.genyleap.com/openproof/api/openapi.yaml is the authoritative machine-readable HTTP contract. This handbook explains the workflow around it.

Part I - Deploy and configure

1. Architecture and trust boundary

A typical production shape is:

```text
Internet
   |
   v
TLS ingress / reverse proxy
   |
   +--> OpenProof (loopback/private listener)
   |      |
   |      +--> PostgreSQL
   |      +--> verification delivery boundary
   |      +--> external identity providers
   |
   +--> your application/API
```

Authentication methods converge into one OpenProof identity:

```text
email/password ---+
passkey -----------+
Google/GitHub -----+
Ethereum/Farcaster +--> OpenProof canonical identity --> OAuth/OIDC --> your product
SAML/LDAP ---------+
```

OpenProof's browser session is for OpenProof account/admin self-service. Product applications should consume OAuth/OIDC tokens; do not copy the OpenProof session cookie into your product domain.

2. Supported packaged hosts

The production installer currently targets:

- Ubuntu 24.04 or newer;
- Debian 13 or newer;
- AMD64 and ARM64;
- systemd-based hosts.

The canonical installer is prebuilt-only. It does not install a compiler, CMake, Boost or a C++ toolchain on the target host. If a matching release bundle is unavailable, installation fails explicitly instead of silently building from source.

3. Prepare before installation

For a real deployment, prepare:

- a public identity hostname such as auth.example.com;
- DNS pointing that hostname to your ingress/server;
- PostgreSQL, either local or external;
- a verification delivery choice: authenticated SMTP, direct Postfix/MX, or your HTTPS delivery webhook;
- TLS strategy: Let's Encrypt, an existing certificate, or external TLS termination;
- external provider credentials only for providers you want to enable immediately.

Reserved/testing hostnames such as *.example.com, *.example.net, *.example.org, *.test, *.invalid and localhost are appropriate for development, not public certificate issuance.

4. Install

Run:

```bash
curl -fsSL https://genyleap.com/install/openproof | sudo sh
```

The bootstrap installer:

1. detects the supported OS and architecture;
2. resolves a release;
3. downloads the matching prebuilt bundle;
4. verifies the release SHA-256 manifest;
5. installs the runtime without compiler/build dependencies;
6. starts the interactive setup wizard.

Package-only installation is also available:

```bash
curl -fsSL https://genyleap.com/install/openproof | sudo sh -s -- --no-setup
sudo openproof setup
```

Pin a version:

```bash
curl -fsSL https://genyleap.com/install/openproof | \
  sudo sh -s -- --version 1.1.0-rc1
```

Use the RC channel explicitly:

```bash
curl -fsSL https://genyleap.com/install/openproof | \
  sudo sh -s -- --channel rc
```

5. Identity settings

The wizard asks for:

- Identity domain - public issuer/origin, e.g. auth.example.com;
- Organization name - operator-facing display name;
- Organization ID - stable tenant identifier;
- Initial owner subject - usually an owner email/login subject.

The identity domain becomes the public issuer:

```text
https://auth.example.com
```

The shared redirect-provider callback becomes:

```text
https://auth.example.com/auth/federated/callback
```

Register that exact callback in upstream provider consoles. Scheme, host, port, path and trailing-slash behavior must match exactly.

Rerunning an incomplete setup

If PostgreSQL was already initialized but setup did not reach its final configured marker, the database is authoritative for the bootstrap organization and initial owner. On rerun OpenProof reconciles the existing state instead of silently replacing it.

A recovery run can therefore report:

```text
Existing deployment detected
Initialized OpenProof identity state found
Organization  Example (example)
Owner subject owner@example.com
```

The existing initial owner is preserved; its password is not recreated merely because setup was rerun.

6. Cryptographic material

The installer generates protected secret files for:

- token/master signing material;
- TOTP credential encryption;
- password pepper;
- recovery-code pepper;
- audit-chain integrity;
- OAuth client-secret protection;
- metrics bearer authentication;
- verification-delivery authentication;
- OIDC RSA signing key.

Generated secrets are not stored in the public repository and use restricted filesystem permissions.

Do not copy these files into source control. Include them in the deployment backup/rotation plan.

7. PostgreSQL

The wizard supports:

```text
1) Install local PostgreSQL
2) Use existing PostgreSQL
```

For a single-host installation, local PostgreSQL is the simplest option. External PostgreSQL is recommended when your infrastructure already provides managed backups, replication or independent database lifecycle.

External URLs must use:

```text
postgres://...
postgresql://...
```

OpenProof applies checksummed migrations before opening the listener.

8. Verification email delivery

OpenProof separates verification issuance from message delivery.

```text
OpenProof
   |
   v
authenticated delivery boundary
   |
   +--> SMTP relay
   +--> local Postfix / direct MX
   +--> your HTTPS webhook
```

Wizard choices:

1. authenticated SMTP relay through the local delivery adapter;
2. direct Postfix/MX;
3. existing HTTPS delivery webhook;
4. configure later.

If you choose direct MX, public email deliverability still requires operator-managed PTR/rDNS, SPF, DKIM, DMARC and reputation controls.

You can edit delivery later:

```bash
sudo openproof config delivery
```

The management command backs up configuration before an edit and restores the previous file when the affected service cannot recover.

9. External sign-in providers

Available setup selections include:

- Google;
- GitHub;
- Microsoft;
- Apple;
- LinkedIn;
- Telegram;
- X;
- Ethereum;
- Farcaster.

Leave the provider list blank if credentials are not ready. Configure later with:

```bash
sudo openproof config providers
```

Common placeholder values such as 0, test, dummy, example and placeholder are treated as deferred configuration rather than production credentials.

Redirect-based providers share the public callback:

```text
https://auth.example.com/auth/federated/callback
```

Provider credentials are mutable after installation; they are not identity database keys.

10. Protected application upstream

OpenProof can protect and proxy configured application routes. Setup asks for:

```text
Upstream host [127.0.0.1]
Upstream port [18080]
Upstream TLS [false]
```

A common single-host deployment is:

```text
OpenProof gateway --> 127.0.0.1:3000 --> product backend
```

Change it later with:

```bash
sudo openproof config main
```

The product backend does not need to be running during the initial OpenProof installation. OpenProof's own identity/OAuth endpoints remain independent of the product upstream.

11. TLS and ingress

Setup supports:

1. Let's Encrypt for a real public DNS name;
2. an existing certificate/key;
3. external ingress / configure TLS later.

For reserved/test domains, interactive setup defaults away from public Let's Encrypt because those names cannot receive a normal public certificate.

The production listener is loopback-oriented. A trusted reverse proxy is expected to overwrite X-Forwarded-For with exactly one validated client IP.

12. Trusted proxy health probes

Production setup enables trusted-proxy client-IP handling. Direct requests to the loopback listener must include one valid X-Forwarded-For value.

Manual local readiness check:

```bash
curl -fsS \
  -H 'X-Forwarded-For: 127.0.0.1' \
  http://127.0.0.1:18443/health/ready
```

Healthy result:

```json
{"status":"ok"}
```

A request sent directly to the loopback listener without the header is intentionally rejected when trusted-proxy mode is enabled.

13. Verify the completed installation

Use:

```bash
sudo openproof status
openproof status --full
sudo openproof doctor
```

A healthy runtime reports:

```text
Binary installed
Configuration present
OpenProof service is running
Local readiness endpoint is healthy
```

Useful operations:

```bash
openproof info
openproof logs --lines 200
openproof logs --follow

sudo openproof start
sudo openproof stop
sudo openproof restart

sudo openproof config main
sudo openproof config providers
sudo openproof config delivery

sudo openproof backup /var/backups/openproof.dump
sudo openproof update
sudo openproof upgrade --channel rc
```

Run openproof doctor after configuration, proxy, DNS or certificate changes.

14. Production checklist

Before public launch confirm:

- real DNS name;
- trusted TLS certificate/ingress;
- PostgreSQL backups and a restore drill;
- real email delivery;
- initial owner protected with MFA/TOTP;
- provider callbacks use the exact production identity origin;

- secrets are outside source control;
- application/client/resource registration is complete;
- OAuth redirect URIs are exact;
- monitoring checks readiness;
- logs and request IDs are retained appropriately;
- token/client/provider key rotation procedures are documented.

Part II - Build with OpenProof

15. Developer mental model

A normal product login is:

```text
1. User selects Sign in
2. Product redirects browser to OpenProof /oauth/authorize
3. OpenProof authenticates the user
4. Provider proof resolves to one canonical OpenProof identity
5. OpenProof redirects back with an authorization code
6. Product exchanges code + PKCE verifier at /oauth/token
7. Product receives access/id/refresh tokens as applicable
8. Product calls APIs with the access token
```

Use OAuth/OIDC for product integration. The OpenProof browser session cookie is not the product application's session mechanism.

16. Discovery and JWKS

OIDC discovery:

```text
https://auth.example.com/.well-known/openid-configuration
```

JWKS:

```text
https://auth.example.com/.well-known/jwks.json
```

Prefer Discovery over hard-coding protocol endpoint paths in reusable clients.

17. Create an application

An active IAL2 owner can use the built-in administration console:

```text
GET /admin/console
```

or the API:

```bash
curl --fail-with-body -b owner.cookies \
  https://auth.example.com/admin/applications \
  -H 'Content-Type: application/json' \
  -d '{
    "identifier":"example-web",
    "name":"Example Web",
    "environment":"production"
  }'
```

Save the returned application identifier.

18. Create an OAuth client

Client kinds:

Kind	Typical use	Client secret
browser	SPA/browser-only application	no
native	mobile/desktop application	no
web	server-side web application	yes
service	machine-to-machine workload	yes

Example:

```bash
curl --fail-with-body -b owner.cookies \
  https://auth.example.com/admin/clients \
  -H 'Content-Type: application/json' \
  -d '{
    "application_id":"APPLICATION_ID",
    "name":"Example Web",
    "kind":"web",
    "redirect_uris":["https://app.example.com/oauth/callback"],
    "scopes":["openid","profile","offline_access"]
  }'
```

Confidential client secrets are returned once. Store them in a backend-only secret store. Never place a web/service client secret in browser JavaScript.

19. Authorization Code + PKCE

Use PKCE S256.

Generate:

- a high-entropy code_verifier;
- code_challenge = BASE64URL(SHA256(code_verifier));
- random state;
- random nonce.

Navigate the browser to:

```text
https://auth.example.com/oauth/authorize
  ?response_type=code
  &client_id=CLIENT_ID
  &redirect_uri=https%3A%2F%2Fapp.example.com%2Foauth%2Fcallback
  &scope=openid%20profile%20offline_access
  &code_challenge=PKCE_CHALLENGE
  &code_challenge_method=S256
  &state=STATE
  &nonce=NONCE
```

At callback, validate the returned state and issuer (iss) before exchanging the code.

20. Exchange the authorization code

```bash
curl --fail-with-body https://auth.example.com/oauth/token \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  --data-urlencode 'grant_type=authorization_code' \
  --data-urlencode 'client_id=CLIENT_ID' \
  --data-urlencode 'code=AUTHORIZATION_CODE' \
  --data-urlencode 'redirect_uri=https://app.example.com/oauth/callback' \
  --data-urlencode 'code_verifier=PKCE_VERIFIER'
```

A successful response can include:

- access_token;
- id_token when openid was requested;
- refresh_token when allowed/requested;
- token type and lifetime metadata.

21. Validate the ID token

Do not merely decode a JWT. Verify the RS256 signature against JWKS and validate at least:

- iss equals the configured OpenProof issuer;
- aud contains/matches your client ID;
- exp has not expired;
- iat is plausible;
- nonce matches the login transaction.

Also validate nbf, azp and at_hash when present/required by your OIDC library.

Use a mature OIDC/JWT library in languages that do not ship an OpenProof SDK.

22. Read current user claims

BASH

```
curl --fail-with-body https://auth.example.com/oauth/userinfo \
  -H 'Authorization: Bearer ACCESS_TOKEN'
```

Your application should key its user record to the stable OpenProof subject (sub), not to an upstream provider email.

23. Refresh tokens

BASH

```
curl --fail-with-body https://auth.example.com/oauth/token \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  --data-urlencode 'grant_type=refresh_token' \
  --data-urlencode 'client_id=CLIENT_ID' \
  --data-urlencode 'refresh_token=REFRESH_TOKEN'
```

Successful refresh rotates the refresh token. Persist the replacement atomically and discard the previous token. Replay protection can revoke the token family.

24. JavaScript/browser SDK

The repository ships a browser-focused JavaScript SDK package named @openproof/identity. It creates state, nonce and PKCE material and verifies callback state/issuer plus the RS256 ID token against JWKS.

JAVASCRIPT

```
import { OpenProofIdentity } from "@openproof/identity";

const identity = new OpenProofIdentity({
  issuer: "https://auth.example.com",
  clientId: "CLIENT_ID",
  redirectUri: "https://app.example.com/oauth/callback",
  scopes: ["openid", "profile", "offline_access"]
});

await identity.login();
```

Callback:

JAVASCRIPT

```
const tokens = await identity.handleCallback();

console.log(tokens.id_token_claims.sub);

const profile = await identity.userInfo(tokens.access_token);
```

Refresh:

JAVASCRIPT

```
const next = await identity.refresh(tokens.refresh_token);
```

The JavaScript SDK is a public-client helper; do not place confidential client secrets in it.

25. Node.js integration

Server-side Node.js may use a standards-compliant OAuth/OIDC library or direct HTTP.

Minimal token exchange:

JAVASCRIPT

```
const body = new URLSearchParams({
  grant_type: "authorization_code",
  client_id: process.env.OPENPROOF_CLIENT_ID,
  code,
  redirect_uri: "https://app.example.com/oauth/callback",
  code_verifier: verifier
});

const response = await fetch("https://auth.example.com/oauth/token", {
  method: "POST",
  headers: {
    "content-type": "application/x-www-form-urlencoded"
  },
  body
});

if (!response.ok) {
  throw new Error(`OpenProof token exchange failed: ${response.status}`);
}

const tokens = await response.json();
```

For a confidential web client, keep its secret on the server only and follow the authentication method advertised by Discovery/current OpenProof documentation. Current OpenProof token endpoints support form-body client credentials for confidential clients rather than HTTP Basic.

For ID tokens, use an OIDC/JWT library to fetch/cache the issuer JWKS and validate signature and claims.

26. PHP integration

OpenProof does not require a PHP-specific SDK. Use a standards-compliant OAuth/OIDC package, or the protocol directly.

Token exchange with cURL:

PHP

```php
<?php

$payload = http_build_query([
    'grant_type' => 'authorization_code',
    'client_id' => getenv('OPENPROOF_CLIENT_ID'),
    'code' => $_GET['code'],
    'redirect_uri' => 'https://app.example.com/oauth/callback',
    'code_verifier' => $_SESSION['openproof_pkce_verifier'],
]);

$curl = curl_init('https://auth.example.com/oauth/token');

curl_setopt_array($curl, [
    CURLOPT_POST => true,
    CURLOPT_POSTFIELDS => $payload,
    CURLOPT_RETURNTRANSFER => true,
    CURLOPT_HTTPHEADER => [
        'Content-Type: application/x-www-form-urlencoded',
    ],
]);

$body = curl_exec($curl);

if ($body === false) {
    throw new RuntimeException(curl_error($curl));
}

$status = curl_getinfo($curl, CURLINFO_RESPONSE_CODE);
curl_close($curl);

if ($status < 200 || $status >= 300) {
    throw new RuntimeException("OpenProof token exchange failed");
}

$tokens = json_decode($body, true, flags: JSON_THROW_ON_ERROR);
```

Use a maintained PHP JWT/OIDC library for ID-token signature and claim validation. Do not implement RSA/JWK verification ad hoc.

27. C++ integration

The repository ships a C++26 module SDK exported as:

CPP

```cpp
import openproof.sdk;
```

Configuration:

CPP

```cpp
auto config = openproof::sdk::ClientConfig::create(
    "https://auth.example.com",
    "CLIENT_ID",
    "http://127.0.0.1:49152/callback",
    {"openid", "profile", "offline_access"});

if (!config) {
    // handle configuration error
}
```

Create a client and authorization transaction:

CPP

```cpp
openproof::sdk::IdentityClient client{std::move(config).value()};

auto login = client.beginLogin();
if (!login) {
    // handle error
}

std::cout << login->authorizationUrl() << '\n';
```

The C++ SDK provides transport-neutral request descriptions for:

- beginLogin();

- authorizationCodeRequest();
- refreshRequest();
- userInfoRequest().

Use your existing HTTP transport such as Boost.Beast, Qt Network or libcurl. A minimal relying-party example is in examples/reference-client.

28. Any other language

Python, Go, Rust, Java, C#, Ruby and other ecosystems can integrate through standard OAuth 2.0/OIDC.

Minimum requirements:

1. load issuer Discovery;
2. generate/store state, nonce and PKCE verifier;
3. redirect to the authorization endpoint;
4. validate callback state and issuer;
5. exchange code at the token endpoint;
6. verify ID-token signature/claims with JWKS;
7. use access tokens for APIs;
8. rotate refresh tokens correctly.

If your framework has a mature OIDC client, configure OpenProof as its issuer instead of writing the protocol by hand.

29. Register a protected API resource

Example:

```bash
curl --fail-with-body -b owner.cookies \
  https://auth.example.com/admin/resources \
  -H 'Content-Type: application/json' \
  -d '{
    "audience":"https://api.example.com/rides",
    "name":"Ride API",
    "scopes":["rides:read","rides:request"]
  }'
```

Request the exact resource/audience and required scopes during authorization when your client needs that API.

The backend should authorize at least:

- token active/valid;
- exact intended audience;
- required scope;
- product-level permission/business rules.

30. Token introspection

A trusted confidential product backend can ask OpenProof whether a token is active:

```bash
curl --fail-with-body https://auth.example.com/oauth/introspect \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  --data-urlencode 'client_id=CONFIDENTIAL_CLIENT_ID' \
  --data-urlencode 'client_secret=CLIENT_SECRET' \
  --data-urlencode 'token=ACCESS_TOKEN'
```

Use introspection when your architecture prefers server-side validation rather than parsing token state locally.

31. Protect routes with the OpenProof gateway

OpenProof can enforce configured route policy before proxying to your backend.

Example configuration:

TOML

```toml
[gateway]
enabled = true
route_prefix = "/"
upstream_host = "127.0.0.1"
upstream_port = 3000
upstream_tls = false

[auth]
enabled = true
provider_id = "local"
organization_id = "example"
protected_route_prefix = "/api"

[[auth.route_policies]]
path_prefix = "/api/reports"
methods = ["GET", "POST"]
required_roles = ["report-reader", "report-admin"]
role_match = "any"
minimum_assurance = "ial2"
required_scope = "reports"
required_audience = "https://api.example.com"
```

Policies are validated before the listener starts. An undeclared protected method/path does not silently fall through as an allow.

32. Service-to-service authentication

Create a service client and provision its allowed audience/scope set. Then use client credentials:

BASH

```bash
curl --fail-with-body https://auth.example.com/oauth/token \
  -H 'Content-Type: application/x-www-form-urlencoded' \
  --data-urlencode 'grant_type=client_credentials' \
  --data-urlencode 'client_id=CLIENT_ID' \
  --data-urlencode 'client_secret=CLIENT_SECRET' \
  --data-urlencode 'scope=rides:dispatch' \
  --data-urlencode 'resource=https://api.example.com/dispatch'
```

The result is an access token only; there is no human session or refresh token.

33. Account signup and verification

When account self-service is enabled and delivery is configured:

BASH

```bash
curl --fail-with-body https://auth.example.com/account/signup \
  -H 'Content-Type: application/json' \
  -d '{
    "email":"user@example.com",
    "password":"REPLACE_WITH_A_LONG_UNIQUE_PASSWORD",
    "display_name":"Example User"
  }'
```

Success returns HTTP 201 with an identity ID and verification_required: true.

Complete the verification using the identifier/secret delivered over the configured email/SMS boundary:

BASH

```bash
curl --fail-with-body https://auth.example.com/account/email/verify \
  -H 'Content-Type: application/json' \
  -d '{
    "verification_id":"VERIFICATION_ID",
    "secret":"ONE_TIME_SECRET"
  }'
```

34. Local password/MFA session flow

The OpenProof local login ceremony is two-step and cookie-bound.

Start:

BASH

```
curl --fail-with-body -c openproof.cookies \
  https://auth.example.com/auth/login \
  -H 'Content-Type: application/json' \
  -d '{"subject":"user@example.com"}'
```

The response contains a transaction/challenge pair. Verify:

BASH

```
curl --fail-with-body -b openproof.cookies -c openproof.cookies \
  https://auth.example.com/auth/mfa/verify \
  -H 'Content-Type: application/json' \
  -d '{
    "transaction_id":"TRANSACTION_ID",
    "challenge_id":"CHALLENGE_ID",
    "password":"PASSWORD",
    "totp":"123456"
  }'
```

A one-time recovery code can replace TOTP while password remains mandatory.

35. Profile, reset and TOTP self-service

Common account endpoints include:

- GET /account/profile;
- PATCH /account/profile;
- POST /account/password/forgot;
- POST /account/password/reset;
- POST /account/email/change;
- POST /account/email/change/verify;
- GET /account/totp;
- POST /account/totp/start;
- POST /account/totp/complete;
- POST /account/totp/disable;
- POST /auth/recovery-codes.

Password-reset initiation is enumeration-safe.

36. Passkeys

Registration:

1. POST /account/passkeys/options;
2. call navigator.credentials.create() with the returned WebAuthn options;
3. POST /account/passkeys with the browser credential;
4. list/remove through the passkey account endpoints.

Login:

1. POST /auth/passkey/options;
2. call navigator.credentials.get();
3. POST /auth/passkey/verify.

Do not transform WebAuthn base64url fields or synthesize authenticator data yourself.

37. Link multiple sign-in methods to one identity

OpenProof does not silently merge accounts by email. After authenticating an existing identity:

TEXT

```
GET /account/connections
```

A redirect-provider connection starts at:

TEXT

```
/account/connections/start?provider=google&return_to=%2F
```

Web3 providers use connection-specific start/complete flows. OpenProof refuses cross-identity attachment and refuses removing the final available sign-in method.

This allows one canonical sub to have, for example:

```text
email/password
+ Google
+ GitHub
+ passkey
+ Ethereum
+ Farcaster
```

without creating multiple unrelated product users.

38. API error contract

Errors use a stable JSON envelope with a code, client-safe message and request ID. Clients should branch on the error code/status, not prose.

Typical semantics:

HTTP	Meaning
400	invalid request
401	authentication missing/invalid
403	authenticated but authority/assurance insufficient
409	state/resource conflict
429	rate limited - back off
5xx	server/dependency failure

Retain the returned request ID for operational correlation.

39. Never log secrets

Do not log:

- passwords;
- TOTP values;
- recovery codes;
- verification secrets;
- authorization codes;
- access tokens;
- refresh tokens;
- confidential client secrets;
- session cookies;
- DPoP proofs;
- private keys.

Secret-bearing responses use no-store semantics where appropriate.

40. Recommended product topology

A clean web architecture is:

TEXT

```
auth.example.com  --> OpenProof

www.example.com   --> product frontend

api.example.com   --> product API
                      |
                      +--> validate/introspect OpenProof access token
                      or
                      +--> sit behind OpenProof gateway
```

Keep the identity origin separate from product state. Your product stores its own business data keyed to the OpenProof subject.

Part III - LLM and MCP integration

41. Machine-readable documentation

Use these canonical machine-readable entry points:

```text
https://docs.genyleap.com/llms.txt
https://docs.genyleap.com/llms-full.txt
https://docs.genyleap.com/openproof/llms.txt
https://docs.genyleap.com/openproof/llms-full.txt
https://docs.genyleap.com/openproof/api/openapi.yaml
```

llms.txt is the compact navigation/index document. llms-full.txt is intended for retrieval/indexing systems that want a larger OpenProof reference in one text resource.

Treat the OpenAPI 3.1 document as the authoritative machine-readable HTTP contract.

42. OpenProof documentation MCP

The public OpenProof docs MCP endpoint is intended to expose public, read-only documentation and schema assistance to compatible AI clients:

```text
https://docs.genyleap.com/openproof/mcp
```

It must never expose deployment secrets, private server files, production credentials or privileged OpenProof administration state.

Read-only MCP capabilities are designed around:

- searching OpenProof documentation;
- retrieving a guide by topic;
- looking up an OpenAPI endpoint;
- generating language-specific integration examples from documented patterns;
- returning installation/production checklists.

Use the normal OpenProof HTTP/OAuth APIs for application actions. The public docs MCP is documentation tooling, not an administrative backdoor into an OpenProof installation.

43. AI-agent safety boundary

When an LLM or agent builds against OpenProof:

- provide it the public handbook/OpenAPI contract;
- do not provide production provider secrets, client secrets, database URLs or private keys;
- do not let a documentation MCP tool mutate an OpenProof deployment;
- require explicit operator approval for separate infrastructure/admin automation;
- prefer typed OpenAPI/OIDC contracts over prose guessing;
- keep secret/token values out of prompts, traces and agent memory.

Part IV - Troubleshooting

44. Configuration validates but service is unhealthy

Run:

```bash
sudo openproof status --full
sudo openproof doctor
openproof logs --lines 200
```

Check the readiness endpoint with the trusted-proxy header when calling loopback directly.

45. Provider login fails

Verify:

1. provider is advertised by GET /auth/providers;
2. client ID/secret are real, not placeholders;
3. registered callback is exactly https://YOUR_IDENTITY_DOMAIN/auth/federated/callback;
4. provider-side application permissions/scopes are enabled;
5. system clock and TLS are valid;
6. correlate the returned request ID with logs.

46. OAuth callback fails

Verify:

- redirect URI is one of the exact registered URIs;
- state matches the transaction;
- returned issuer matches the configured issuer;
- PKCE verifier matches the challenge;
- authorization code has not been reused/expired;
- confidential credentials were not accidentally put in browser code.

47. API returns 401 vs 403

- 401: no valid authentication/token/session.
- 403: identity is authenticated but lacks membership, role, assurance, scope or audience required by policy.

Do not solve 403 by weakening authentication checks until the failing policy dimension is understood.

48. SMTP test configuration

A hostname such as mail.example.com is documentation-only. A running delivery adapter does not prove that a placeholder SMTP host can deliver mail. Before signup/recovery testing, configure a real SMTP relay or webhook.

49. Product upstream unavailable

OpenProof identity/OAuth endpoints can remain healthy while the configured product upstream is absent. Gateway-proxied application routes fail until that upstream is listening and reachable.

Reference links

- Documentation: https://docs.genyleap.com/openproof/
- Deployment & Developer Handbook: https://docs.genyleap.com/openproof/handbook
- Developer/API guide: https://docs.genyleap.com/openproof/develop
- AI & MCP: https://docs.genyleap.com/openproof/ai
- Interactive API reference: https://docs.genyleap.com/openproof/api/
- OpenAPI 3.1: https://docs.genyleap.com/openproof/api/openapi.yaml
- Source: https://github.com/genyleap/openproof

Repository references:

- docs/INSTALLATION.md
- docs/CONFIGURATION.md
- docs/API_GUIDE.md
- docs/PROVIDER_SETUP.md
- docs/OPERATIONS.md
- docs/openapi.yaml
- sdk/javascript/
- sdk/cpp/
- examples/reference-client/