Manual completo das ferramentas Lastrium

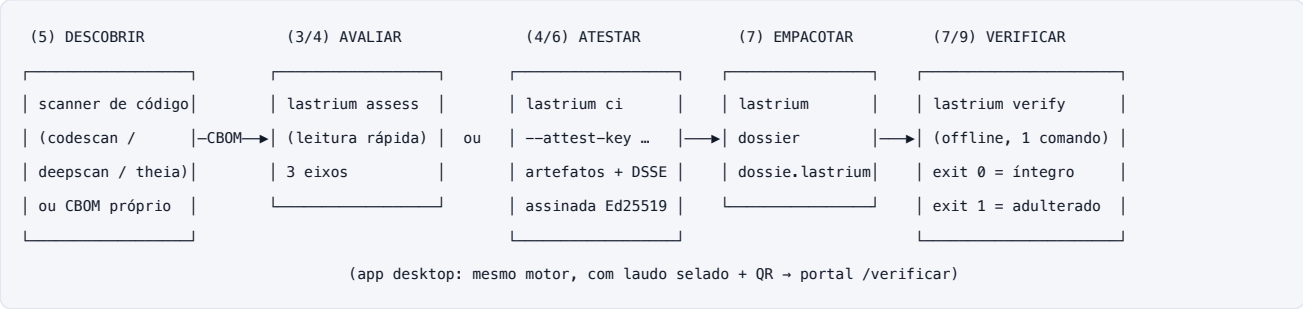
Instalação, uso e validação de cada ferramenta, passo a passo. Público: pessoa técnica que nunca viu o produto (CISO hands-on, analista de segurança, auditor técnico). Versão do motor coberta: **0.3.3**. Última revisão: 2026-07-04.

> **Postura honesta (vale para todo o manual):** o Lastrium entrega **evidência de postura criptográfica**, não certificação legal — a responsabilidade regulatória é, por norma, da instituição. E a regra inegociável do classificador: **algoritmo não reconhecido é `indeterminado`, nunca `quantum-seguro`**. Em nenhum ponto deste manual (nem do produto) um "não sei" vira um "está seguro".

Mapa: qual ferramenta para qual tarefa

QUERO...	FERRAMENTA	SEÇÃO
Avaliar um inventário e gerar um laudo, **sem terminal**	App desktop (macOS/Linux)	1
Instalar a CLI e confirmar que o motor está são	CLI `lastrium` + `healthcheck`	2
Avaliação rápida de um CBOM (relatório em segundos)	`lastrium assess`	3
Rodar o pipeline completo na esteira, com atestação assinada	`lastrium ci`	4
Descobrir a criptografia no código/filesystem (gerar o CBOM)	Scanners: codescan / deepscan / cbomkit-theia	5
Atestar a postura de um artefato/imagem com cosign	`lastrium attest`	6
Entregar a prova ao auditor e deixá-lo conferir sozinho	`lastrium dossier` + `lastrium verify`	7
Checar revogação de certificados (OCSP/CRL) com evidência da CA	`lastrium revocation`	8
Baixar instaladores, ativar licença, verificar um selo	Portal (web)	9
Ver tudo funcionando de ponta a ponta num banco fictício	Laboratório Banco Meridiano	10
Provar o produto inteiro em ~10 minutos	Roteiro final	11

O fluxo, num diagrama



Os **três eixos** de toda avaliação (app e CLI) — cada um com grau de prova distinto:

1. **Criptográfico** — *verificado* do inventário: exposição quântica de cada algoritmo (Shor/Grover), priorização de migração (Mosca/HNDL).

2. **Custódia** — *detectado* do inventário e do filesystem: certificado vencido/revogado, chave comprometida, postura de guarda (texto claro, permissão) — nunca lendo o valor do segredo.
3. **Governança** — *autodeclarado e selado*: deveres organizacionais respondidos pela instituição, com documentos anexados e hasheados no dossiê. O Lastrium sela a declaração; não afirma que ela é verdadeira.

Convenções deste manual

- Cada passo mostra **o que o comando faz** antes do bloco de código e **o que você deve ver** depois.
- Comandos foram executados e as saídas conferidas contra o motor 0.3.3. Onde um detalhe não pôde ser confirmado nas fontes, ele está marcado com **[confirmar]**.
- `./lastrium` = binário compilado no diretório atual; se você moveu para o `PATH`, use apenas `lastrium`.

---

1. App desktop

1.1 O que é e quando usar

O caminho do usuário final: um aplicativo de mesa (macOS/Linux) em que você arrasta um arquivo CBOM, vê a avaliação nos três eixos e gera um **Laudo de Postura Criptográfica selado** (PDF com hash SHA-256 + QR verificável no portal). Use quando quiser avaliar e documentar sem terminal — a esteira de CI e o dossiê completo são papel da CLI (seção 4).

1.2 Pré-requisitos

- **macOS**: Apple Silicon (o instalador é `aarch64`). O app é **assinado (Developer ID) e notarizado pela Apple** — abre sem aviso do Gatekeeper.
- **Linux**: x86_64, com `.deb` (Debian/Ubuntu/Mint) ou `.rpm` (Fedora/RHEL/openSUSE).
- Conta na **Área do Cliente** (https://lastrium.com.br/area) para baixar o instalador e, para o laudo **selado**, uma **licença ativa**. Sem licença o app roda em **modo trial**: avaliação completa, mas sem selar o laudo.
- Um CBOM para testar. Se não tiver, baixe o exemplo público (core banking fictício, 52 ativos): https://lastrium.com.br/exemplos/cbom-meridiano-grande.json

1.3 Instalação passo a passo

macOS (Apple Silicon):

1. Acesse **https://lastrium.com.br/area**, clique em **Cadastrar** e crie a conta (e-mail, senha, nome). Você recebe um e-mail de boas-vindas com o link de download.
2. Na Área do Cliente, baixe `Lastrium_<versão>_aarch64.dmg`.
3. Abra o `.dmg` e arraste **Lastrium** para a pasta **Aplicativos**.
4. Dê duplo clique para abrir. **Você deve ver:** o app abrindo normalmente, **sem** aviso de segurança ("app não confiável") — ele é assinado e notarizado.

Linux (x86_64): baixe o pacote da sua distribuição na Área do Cliente e instale. O que este passo faz: instala o app e resolve as dependências GTK/WebKit automaticamente.

```
# Debian/Ubuntu (.deb)
sudo apt install ./Lastrium_*_amd64.deb

# Fedora/RHEL/openSUSE (.rpm)
sudo dnf install ./Lastrium-*.x86_64.rpm
```

Você deve ver: o gerenciador de pacotes concluindo sem erro e o **Lastrium** disponível no menu de aplicativos.

> A pipeline de release também produz um `.AppImage` para Linux (artefato do CI em `desktop/src-tauri/target/release/bundle/appimage/`). A disponibilidade do `.AppImage` na Área do Cliente **[confirmar]** — o material publicado documenta os caminhos `.deb` / `.rpm`.

Ativar a licença (para o laudo selado):

1. Na primeira execução, o app mostra o **HWID** (identificador da máquina) e o estado da licença.
2. Copie o HWID e informe-o na Área do Cliente (campo *HWID*).
3. A equipe Lastrium emite a licença para esse HWID e você recebe a **chave**.
4. No app, clique em **Ativar licença**, cole a chave e confirme. **Você deve ver:** o topo do app mostrando *"Licenciado · |<seu nome>"*.

1.4 Uso passo a passo

1. **Carregar o inventário.** O que este passo faz: entrega ao motor embarcado (o mesmo `lastrium` da CLI, rodando como *sidecar*) o CBOM CycloneDX 1.6 a avaliar. **Arraste** o arquivo (ex.: `cbom-meridiano-grande.json`) para a janela — ou clique para escolher. **Você deve ver:** a avaliação aparecendo em segundos.
2. **Ler os três eixos. Você deve ver:** contagem por status (vulnerável / enfraquecido / indeterminado / seguro), severidade e gráficos; a **análise jurídica** por ativo (CMN 5.274 / BCB 538 / LGPD); o bloco **Custódia — a tratar** (certificados/chaves com problema, dispositivo normativo e severidade); e, se houver um `.lastrium-governanca.yml` ao lado do CBOM, o bloco **Governança — autodeclarado** com o chip **⛓ do anexo selado** (nome + hash).
3. **Exportar (opcional).** PDF (executivo/técnico com os três eixos), TXT ou JSON.
4. **Gerar o Laudo selado.** O que este passo faz: monta o documento jurídico, calcula o **hash SHA-256** do laudo, obtém um **selo** do portal (**só o hash trafega** — nunca o conteúdo) e embute um **QR** no PDF. Clique em **Gerar Laudo** → preencha instituição, destinatário, responsável técnico, conclusão (vem pré-preenchida) e observações → **Gerar laudo selado** → salve o PDF. **Você deve ver:** na última página do PDF, a impressão digital (SHA-256), o identificador do selo e o QR code.

1.5 Validação detalhada

A prova de que o laudo é autêntico é externa ao app — qualquer pessoa a executa:

1. Acesse **https://licensing.lastrium.com.br/verificar**.
2. Escaneie o QR do laudo (ou cole o conteúdo lido) e clique **Verificar**.
3. **Você deve ver: "✓ Selo AUTÊNTICO e ÍNTEGRO"**, com o emissor, a data e o hash.
4. **Confira** que o hash exibido no portal é **idêntico** ao impresso na última página do laudo. Hash igual = o PDF que você tem em mãos é o que foi selado.

Teste negativo: cole no verificador um conteúdo de QR de **outro** laudo (ou um texto qualquer). **Você deve ver** o selo **não** validar — um selo de outro ambiente/chave não passa. Quem lê o QR sem o portal vê apenas texto cifrado.

Checklist de aceite do app:

#	O QUE VALIDAR	RESULTADO ESPERADO
1	App abre	macOS: duplo clique sem aviso (notarizado). Linux: instala e abre pelo menu
2	Avaliação de um CBOM	contagem por status + análise jurídica por ativo
3	Laudo selado	PDF com hash SHA-256 + QR
4	Verificação do selo	"AUTÊNTICO e ÍNTEGRO", hash bate com o impresso

1.6 Problemas comuns

- **App "danificado" no macOS** — não deve mais ocorrer (builds notarizadas desde a 0.3.2). Se acontecer numa build antiga: `xattr -dr com.apple.quarantine /Applications/Lastrium.app`.
- **Dependência faltando no Linux** — prefira `apt install ./…deb` / `dnf install ./…rpm` (resolvem GTK/WebKit sozinhos). Se instalou com `dpkg -i`, rode depois `sudo apt -f install`.

- **"Licença necessária" ao selar** — o laudo *selado* exige licença ativa (seção 1.3); o laudo *sem selo* pode ser gerado offline.
- **Selo "não validado" no portal** — confirme que colou o conteúdo **completo** do QR; um selo emitido por outra chave/ambiente não valida.

---

2. CLI `lastrium` — instalação e `healthcheck`

2.1 O que é e quando usar

`lastrium` é o motor em um único binário estático (sem dependências): os subcomandos `assess`, `ci`, `attest`, `healthcheck`, `verify`, `dossier`, `revocation` e `license`. Use na máquina do analista ou na esteira de CI — tudo offline por padrão.

2.2 Pré-requisitos

- **Via download:** nenhum — é um binário único por plataforma.
- **Via código-fonte: Go 1.25+** (o modo com OPA/Rego exige 1.25; não retroceder).

2.3 Instalação — modo A: download (Área do Cliente)

O que este passo faz: baixa o binário pronto e o torna executável.

1. Na **Área do Cliente** (https://lastrium.com.br/area), baixe o binário da sua plataforma: `lastrium-<versão>-<os>-<arch>` (Linux x86_64/ARM64, macOS, Windows).
2. Dê permissão de execução e, opcionalmente, mova para o `PATH`:

```
chmod +x lastrium-*
sudo mv lastrium-* /usr/local/bin/lastrium
```

Você deve ver: `lastrium healthcheck` respondendo (passo 2.5).

2.4 Instalação — modo B: compilar do código-fonte

O que este passo faz: clona o repositório do motor e compila. Há **dois sabores** de build:

- **Sem tag (stdlib-puro, ~4 MB)** — é o binário público: `assess`, `ci` simples, `verify`, `dossier` etc. O gate de política roda na implementação Go.
- **Com `-tags rego` (~29 MB)** — embute o OPA v1.18; o gate de política roda em Rego (`policy/lastrium.rego`). O resultado do gate é equivalente — a tag só confina o OPA no binário.

```
git clone <repo-do-engine> lastrium && cd lastrium
make test             # roda a suíte de testes (deve passar)
make build            # gera ./lastrium (stdlib-puro, ~4 MB)

# equivalente manual:
go build -o lastrium ./cmd/lastrium

# sabor com OPA/Rego:
go test -tags rego ./...
go build -tags rego -o lastrium ./cmd/lastrium
```

Você deve ver: `make test` terminando com todos os pacotes `ok`, e o arquivo `./lastrium` criado.

Outros alvos úteis do `Makefile` (raiz):

ALVO	O QUE FAZ
`make run`	build + `assess` no `testdata/sample-cbom.json`
`make demo`	avaliação no CBOM grande de exemplo (`testdata/cbom-meridiano-grande.json`, 52 ativos)
`make dist`	binários públicos multiplataforma em `dist/` (CGO off, `-trimpath`, 5 alvos: linux/amd64, linux/arm64, darwin/amd64, darwin/arm64, windows/amd64)
`make sbom`	SBOM CycloneDX das dependências em `dist/lastrium-sbom.cdx.json`
`make dogfood`	o Lastrium avalia a própria criptografia (`testdata/cbom-lastrium-self.json`)
`make vet`	`go vet ./...`
`make codescan-image` / `make deepscan-image`	imagens dos scanners (seção 5)
`make clean`	remove `./lastrium` e `dist/`

> Ambiente offline/air-gap para o build: `GOPROXY=off GOFLAGS=-mod=mod`.

2.5 Uso: `lastrium healthcheck` — o self-test do motor

O que este comando faz: um autoteste mínimo e **determinístico** do núcleo (o classificador de exposição quântica), sem rede, sem arquivos externos e sem efeitos colaterais. Ele confere **duas invariantes**:

1. **RSA é classificado como quebrável por Shor** (`quantum-vulneravel`) — se isso falhar, o classificador está degradado.
2. **Algoritmo desconhecido NUNCA é seguro** — um nome inventado (`Xyzzy-Cipher-404`) tem de sair `indeterminado`.

```
./lastrium healthcheck
echo $?
```

Você deve ver:

```
ok lastrium 0.3.3
0
```

- **exit 0** = saudável (`ok lastrium <versão>` no stdout).
- **exit 1** = degradado — com a causa no stderr: `healthcheck: classificador degradado (RSA não classificado como vulnerável)` ou `healthcheck: invariante 'desconhecido → indeterminado' violada`.

O comando não tem flags. Ele serve também de `HEALTHCHECK` da imagem Docker do motor (distroless, sem shell) e de probe de liveness/readiness:

```
HEALTHCHECK CMD ["/usr/local/bin/lastrium", "healthcheck"]
```

2.6 Validação detalhada

1. `./lastrium healthcheck` → stdout `ok lastrium 0.3.3`, **exit 0**.
2. `./lastrium` sem argumentos → mensagem de uso `uso: lastrium <assess|ci|verify> [flags]` no stderr, **exit 1** (é o comportamento esperado, não um defeito).
3. Se compilou do fonte: `make test` inteiro verde — inclui o teste da regra inegociável (`TestUnknownIsNeverSafe`) e o de que o probe de chaves nunca lê o segredo (`TestAnnotate_NuncaVazaSegredo`).

2.7 Problemas comuns

- **`go build` reclama da versão do Go** — o projeto exige Go 1.25 (elevado pelo OPA v1.18). Atualize o toolchain; não force downgrade.

- **Build em ambiente sem internet** — use `GOPROXY=off GOFLAGS=-mod=mod` (o `Makefile` já exporta `GOFLAGS=-mod=mod`).
- **Aviso `[AVISO] lastrium: chave de licença de TESTE embarcada — NÃO usar em produção`** — o binário foi compilado sem a pubkey de produção (`LASTRIUM_PUBLIC_HEX` no `make dist`). Para avaliar o produto, é inofensivo; para produção, use o binário oficial da Área do Cliente.

---

3. `lastrium assess` — avaliação rápida de um CBOM

3.1 O que é e quando usar

Lê um CBOM CycloneDX (1.4–1.7, gerado pelos scanners abertos da PQCA ou pelos do Lastrium) e imprime a avaliação nos três eixos em segundos. Use para a primeira leitura de um inventário ou para inspeção pontual — sem atestação (isso é o `ci`, seção 4).

3.2 Pré-requisitos

- CLI instalada (seção 2).
- Um CBOM. Exemplos que acompanham o repositório: `testdata/sample-cbom.json` (9 ativos) e `testdata/cbom-meridiano-grande.json` (52 ativos). Público: https://lastrium.com.br/exemplos/cbom-meridiano-grande.json

3.3 Flags (todas — confirmadas no código)

FLAG	SIGNIFICADO
`--json`	emitir resultado em JSON (para máquina) em vez do texto do auditor
`--as-of YYYY-MM-DD`	data de avaliação p/ vencimento de certificados; vazio desabilita. O "hoje" é **input** — o motor nunca lê o relógio (determinismo)
`--probe-root <dir>`	raiz do filesystem para inspecionar a **postura de guarda** das chaves (permissão + texto claro/cifrado); **nunca lê o valor** da chave

3.4 Uso passo a passo

1. **Avaliação básica.** O que este passo faz: roda o pipeline FileScanner → QuantumEnricher e imprime o relatório legível + o eixo de custódia.

```
./lastrium assess testdata/sample-cbom.json
```

Você deve ver (trecho real, motor 0.3.3):

```
LASTRIUM — avaliacao de postura criptografica (exposicao quantica)
================================================================
Ativos criptograficos analisados: 9

* RSA            [src/auth/jwt_signer.go:88]
    status: quantum-vulneravel     severidade: critico     ameaca: Shor
    RSA: fatoracao resolvida em tempo polinomial por Shor
...
* ML-KEM-768     [src/transport/pqc.go:30]
    status: quantum-seguro         severidade: info
...
* Obfusca-XOR    [src/internal/scramble.go:5]
    status: indeterminado          severidade: desconhecido
    algoritmo nao reconhecido (primitivo "cipher") — requer revisao manual; nao classificar como seguro

Resumo
------
  indeterminado            1
  quantum-enfraquecido     5
  quantum-seguro           1
  quantum-vulneravel       2

  [!] 1 ativo(s) indeterminado(s): requerem revisao manual — NAO classificados como seguros.
  [!] Score HNDL incompleto: falta input de sensibilidade/retencao por servico.
```

2. **Saída JSON.** O que este passo faz: mesma avaliação, serializada para integração.

```
./lastrium assess testdata/sample-cbom.json --json
```

Você deve ver: um JSON com as chaves de topo `enriquecido` (eixo quântico) e `custodia`; se houver `.lastrium-governanca.yml` ao lado do CBOM, também `governanca`.

3. **Com data de avaliação (vencimento de certificados).** O que este passo faz: habilita a checagem determinística de validade de certificados contra a data que **você** fornece (janela de "expira em breve" = 30 dias).

```
./lastrium assess testdata/cbom-meridiano-grande.json --as-of 2026-07-04
```

4. **Com inspeção de guarda de chaves.** O que este passo faz: para cada chave do inventário localizável sob `--probe-root`, anota a permissão do arquivo (`lastrium:file_mode`) e se está em texto claro/cifrada (`lastrium:storage`), lendo apenas `os.Stat` + o prefixo do cabeçalho PEM/OpenSSH — **jamais o valor do segredo**.

```
./lastrium assess inventario.cbom.json --probe-root ./repo
```

3.5 Validação detalhada

1. **Exit code**: `0` = avaliação emitida; `1` = erro (arquivo inexistente, CBOM inválido, flag malformada — a mensagem sai como `lastrium: …` no stderr).
2. **A regra "indeterminado nunca é seguro" em ação**: no `sample-cbom.json`, o ativo `Obfusca-XOR` (nome inventado) tem de aparecer como `indeterminado / severidade: desconhecido`, e o rodapé tem de trazer o aviso `NAO classificados como seguros`. Se algum dia um desconhecido aparecer como `quantum-seguro`, o motor está quebrado (é exatamente o que o `healthcheck` vigia).
3. **Determinismo**: rode o mesmo comando duas vezes e compare — `./lastrium assess testdata/sample-cbom.json --json | shasum -a 256` deve dar o **mesmo hash** nas duas execuções (não há relógio nem aleatoriedade).
4. **Teste negativo**: `./lastrium assess arquivo-que-nao-existe.json` → **exit 1** com erro no stderr.

5. **`--as-of` inválido**: `--as-of 04/07/2026` → erro `--as-of "04/07/2026" inválido: use YYYY-MM-DD`, exit 1.

3.6 Problemas comuns

- **"Score HNDL incompleto"** — não é erro: o plano de migração por risco temporal (teorema de Mosca) exige a classificação de sensibilidade/retenção dos dados (`hndl.data_classification` no `.lastrium.yml`, usada pelo `ci`). Sem ela o motor marca **incompleto** — nunca presume "ok".
- **CBOM com `specVersion` fora da faixa testada (1.4–1.7)** — o motor lê best-effort e emite **aviso não-fatal** no stderr; não bloqueia.
- **Governança não aparece** — o eixo é opt-in: exige `.lastrium-governanca.yml` **ao lado do arquivo CBOM**.

---

4. `lastrium ci` — o pipeline completo da esteira

4.1 O que é e quando usar

O contrato que a esteira de CI/CD chama: **descobre/ingere → avalia → mapeia à norma → sela**. Produz todos os artefatos de prova num diretório e a **atestação DSSE assinada (Ed25519)** que os cobre. Use a cada build/release ou em execuções periódicas de postura; é o insumo do dossiê (seção 7).

4.2 Pré-requisitos

- CLI instalada (seção 2).
- Um CBOM no diretório `--source` (ou um scanner containerizado — seção 5). O `ci` procura o CBOM nesta ordem: `<source>/cbom.cdx.json` → `<source>/.lastrium/cbom.json` → o próprio `--source` se terminar em `.json`.
- `openssl` para gerar o par de chaves de atestação (uma vez).
- Opcional: `.lastrium.yml` na raiz do repositório (política, formatos, HNDL) e `.lastrium-governanca.yml` (governança autodeclarada).
- Opcional: token de licença (`--license` ou env `LASTRIUM_LICENSE`). Sem licença o `ci` roda em modo avaliação, com aviso no stderr.

4.3 Flags (todas — confirmadas no código; precedência: flag > config > default)

FLAG	DEFAULT	SIGNIFICADO
`--source`	`.`	diretório raiz do repositório a escanear
`--config`	`./.lastrium.yml`	caminho do arquivo de configuração
`--out`	`./lastrium-out`	diretório de saída dos artefatos
`--format`	`cyclonedx,sarif,gitlab,json,md`	formatos a emitir (csv)
`--policy-mode`	`warn`	`warn` | `fail` (sobrepõe a config)
`--fail-on`	`shor-breakable,critical`	tokens de falha (csv): `shor-breakable`, `critical`, `indeterminate`, `grover`
`--baseline`	—	caminho de um `summary.json` anterior (achados já conhecidos não disparam o gate)
`--scanner-image`	—	ref@digest da imagem do scanner containerizado (seção 5)
`--scanner-runtime`	`docker`	runtime do container (`docker` | `podman`)
`--scanner-args`	`dir,/src`	args do entrypoint do scanner (csv; default = contrato cbomkit-theia)
`--scanner-allow-network`	`false`	libera a REDE do container do scanner (risco de egresso do código; só com imagem confiável)
`--scanner-timeout`	`0` (= 120 s)	timeout do container em segundos (modo profundo precisa de ~300)
`--license`	env `LASTRIUM_LICENSE`	token de licença Lastrium
`--offline`	`false`	modo offline (sem chamadas de rede)
`--attest-key`	env `LASTRIUM_ATTEST_KEY`	chave privada Ed25519 PEM para assinar a atestação
`--probe-root`	`--source` (se diretório)	raiz p/ inspecionar a postura de guarda das chaves; nunca lê o valor
`--rekor-url`	—	**[reservado]** Rekor não implementado; aceita e ignora com aviso

Significado dos tokens de `--fail-on`: `shor-breakable` → status `quantum-vulneravel`; `critical` → severidade `critico`; `indeterminate` → status `indeterminado`; `grover` → status `quantum-enfraquecido`. Um ativo que case com **qualquer** token dispara o gate.

Exit codes (contrato):

EXIT	SIGNIFICADO
`0`	sem violação, **ou** violação em modo `warn` (artefatos sempre emitidos)
`1`	violação de política em modo `fail`
`2`	erro operacional (scanner/config/IO)
`3`	licença inválida/expirada

4.4 Uso passo a passo

1. **Gerar o par de chaves Ed25519 (uma única vez).** O que este passo faz: cria a chave privada (PKCS8 PEM) que assina a atestação e a pública (PKIX PEM) que o auditor usa para verificar. Guarde a privada como segredo do CI; distribua a pública.

```
openssl genpkey -algorithm ed25519 -out attest.key
openssl pkey -in attest.key -pubout -out attest.pub
```

Você deve ver: os dois arquivos criados; `attest.key` começa com `-----BEGIN PRIVATE KEY-----` e `attest.pub` com `-----BEGIN PUBLIC KEY-----`.

2. **Preparar a origem.** O que este passo faz: coloca um CBOM onde o `ci` o encontra (aqui usamos o exemplo do repositório; em produção seria o CBOM do seu scanner, ou um `--scanner-image`).

```
mkdir -p src && cp testdata/cbom-meridiano-grande.json src/cbom.cdx.json
```

3. **Rodar o pipeline assinado e determinístico.** O que este passo faz: avalia o CBOM, mapeia à norma (RegPack `bacen-br`), aplica o gate de política, gera todos os artefatos e os **assina** num envelope DSSE. `SOURCE_DATE_EPOCH` fixa o timestamp — é o que torna a atestação reproduzível byte-a-byte.

```
SOURCE_DATE_EPOCH=$(date +%s) ./lastrium ci --source src --out out \
  --attest-key attest.key --offline
echo "exit=$?"
ls -1 out/
```

Você deve ver (saída real; os avisos aparecem só sem licença de produção):

```
[AVISO] lastrium: chave de licença de TESTE embarcada — NÃO usar em produção
[AVISO] lastrium: rodando SEM licença (modo avaliação)
exit=0
attestation.dsse.json
cbom.cdx.json
conformidade-bcb538.json
custodia.json
gap-report.json
gl-sast-report.json
lastrium.sarif
report.md
summary.json
```

Alternativas equivalentes de assinatura: `LASTRIUM_ATTEST_KEY=attest.key lastrium ci --source . --out ./out` (via env). **Sem chave nenhuma**, o envelope sai com `signatures: []` + aviso no stderr (exit 0) — os artefatos existem, mas não provam autoria.

4. **Entender cada artefato gerado** (todos cobertos pela atestação):

ARTEFATO	O QUE É
`cbom.cdx.json`	o CBOM avaliado, reemitido em **passthrough fiel** (bytes originais preservados)
`gap-report.json`	lacunas regulatórias por ativo (mapeamento norma↔achado do RegPack)
`custodia.json`	violações **detectadas** de custódia (art. 3º, §12/3º-A): revogado, vencido, chave comprometida, guarda exposta
`conformidade-bcb538.json`	**Relatório de Conformidade BCB 538 por dispositivo** — ver 4.5
`gap-governanca.json`	eixo de governança autodeclarado — **só existe** se houver `.lastrium-governanca.yml` com respostas; os documentos citados entram como artefatos `evidencia--<pergunta>--<arquivo>`, hasheados e selados
`report.md`	relatório técnico legível (por ativo + seção de conformidade por dispositivo)
`summary.json`	resumo para máquina: `gate_result`, contagens, `versions` (engine/pack), `asset_keys` (para baseline), plano de migração, sumário de conformidade
`lastrium.sarif`	achados no formato SARIF (GitHub code scanning etc.)
`gl-sast-report.json`	achados no formato GitLab SAST
`attestation.dsse.json`	o envelope **DSSE assinado** (predicate `https://lastrium.com.br/attestations/cryptoposture/v1`), cujos *subjects* são os SHA-256 de todos os artefatos acima

5. **Provar o determinismo (rodar 2× → byte-a-byte).** O que este passo faz: repete o `ci` com o **mesmo** epoch e compara os hashes — mesma chave + mesmo `SOURCE_DATE_EPOCH` + mesma entrada ⇒ `attestation.dsse.json` idêntico.

```
SOURCE_DATE_EPOCH=1750000000 ./lastrium ci --source src --out out  --attest-key attest.key --offline
SOURCE_DATE_EPOCH=1750000000 ./lastrium ci --source src --out out2 --attest-key attest.key --offline
shasum -a 256 out/attestation.dsse.json out2/attestation.dsse.json
```

Você deve ver: os **dois hashes idênticos**.

6. **Gate de política em modo `fail`.** O que este passo faz: transforma o achado em reprovação da esteira (exit ≠ 0).

```
./lastrium ci --source src --out out3 --policy-mode fail --fail-on shor-breakable --offline
echo "exit=$?"
```

Você deve ver: `exit=1` (o CBOM de exemplo tem ativos quebráveis por Shor) e, dentro de `out3/summary.json`, `"gate_result": "fail"`. Os artefatos são emitidos mesmo assim — a esteira reprova, mas a evidência fica.

4.5 `conformidade-bcb538.json` — o relatório por dispositivo

É a visão **artigo-a-artigo** da Res. CMN 5.274 / BCB 538: para cada **dispositivo normativo** do pacote regulatório (ex.: `art. 3º, §12, I (c/c §7º)`), o relatório agrega as evidências dos três eixos e calcula um status. Todo dispositivo do RegPack aparece — mesmo sem achado (checklist completo, nada some em silêncio). **Os 5 status possíveis:**

STATUS	SIGNIFICADO
`atende`	evidência positiva sob o dispositivo (ex.: só ativos pós-quânticos)
`nao-atende`	gap técnico (vulnerável/enfraquecido) **ou** violação de custódia. Com prazo regulatório vencido, marca também `nao_conformidade_dupla`
`indeterminado`	algoritmo não reconhecido sob o dispositivo — nunca presumido seguro
`sem-achado-aplicavel`	o dispositivo consta (checklist completo), mas nenhum ativo o aciona
`pendente-validacao-juridica`	**todos** os controles do dispositivo têm vinculação `provisorio` no RegPack — conformidade não é afirmada **nem negada**; falta o jurista confirmar o vínculo norma↔achado

Regras honestas embutidas: a governança autodeclarada entra como **nota ao lado** do status técnico e **nunca o altera** (o técnico prevalece); o motor **não inventa dispositivo** fora do pacote (classes sem controle mapeado saem listadas em `sem_controle_mapeado`); é leitura **estruturada** do RegPack, não parecer jurídico.

O sumário (`sumario`) traz a contagem por status + `nao_conformidade_dupla`. Exemplo real (CBOM de 52 ativos, pack `bacen-br@2026.06-go`):

```
"sumario": {"atende": 0, "nao_atende": 6, "indeterminado": 0,
            "sem_achado_aplicavel": 0, "pendente_validacao_juridica": 4,
            "nao_conformidade_dupla": 0}
```

4.6 Validação detalhada

1. **Exit code correto**: `echo $?` após o `ci` — `0` (warn), `1` (fail com violação), `2` (ex.: `--source` sem CBOM → mensagem `CBOM nao encontrado em …` no stderr), `3` (licença inválida — diagnostique o token com `lastrium license verify <token>`).
2. **Artefatos completos**: `ls out/` deve listar os arquivos da tabela 4.4 (9 arquivos sem governança; + `gap-governanca.json` e anexos `evidencia--…` com governança).
3. **A atestação assinou tudo**: rode `./lastrium verify out/attestation.dsse.json --key attest.pub --artifacts-dir out/` → **você deve ver** uma linha `OK <artefato> (<hash>…)` para **cada** arquivo + `todos os artefatos íntegros`, exit 0 (detalhes na seção 7).
4. **Determinismo**: passo 5 acima — hashes idênticos.

5. `summary.json` **coerente**: com o CBOM de exemplo, os campos reais são `"gate_result": "warn"`, `"total_assets": 52`, `"vulnerable_count": 21`, `"indeterminate_count": 2`, `"unique_vulnerable_algorithms": 7`, `"versions": {"engine": "0.3.3", "pack": "bacen-br@2026.06-go"}` (o sufixo `-go` indica o gate Go; com `-tags rego`, `bacen-br@2026.06`).
6. **Teste negativo do gate**: passo 6 acima — `--policy-mode fail` → exit 1.

4.7 Problemas comuns

- `CBOM nao encontrado` (exit 2) — o `--source` não tem `cbom.cdx.json` nem `.lastrium/cbom.json`, e não aponta para um `.json`. Copie o CBOM para `<source>/cbom.cdx.json` ou use `--scanner-image`.
- `rodando SEM licença (modo avaliação)` — informe `--license` / `LASTRIUM_LICENSE` com um token válido; o campo `licensed` do `summary.json` vira `true`.
- `--rekor-url` **não faz nada** — correto: a âncora no log de transparência Rekor não está implementada (exige rede; contraria o offline-first). A flag é aceita e ignorada com aviso.
- **Plano de migração (Mosca) ausente/incompleto** — configure `hndl.assessment_year` e `hndl.data_classification` no `.lastrium.yml` (há um exemplo completo em `lab/.lastrium.yml`); sem classificação o item sai `incompleto`, nunca "ok".

---

5. Scanners de descoberta — de onde vem o CBOM

5.1 O que é e quando usar

O `ci` ingere um CBOM pronto (air-gap) **ou gera na hora**, apontando `--scanner-image` para um scanner containerizado. O contrato é único: o container recebe o código montado em `/src` **(somente-leitura)** e emite um **CBOM CycloneDX 1.6 no stdout** — o scanner é uma peça trocável; o valor está na avaliação e na prova. Use o **codescan** para varrer código-fonte offline, o **deepscan** para análise semântica profunda de Java, o **cbomkit-theia** para o filesystem (certificados/chaves/configs) — ou traga seu próprio CBOM.

5.2 Pré-requisitos

- **Docker** (ou Podman, via `--scanner-runtime podman`).
- Para construir as imagens a partir do repositório do motor: Go (codescan) e rede (deepscan baixa SonarQube + plugin).

5.3 `lastrium-codescan` — o scanner leve (offline)

Scanner heurístico de **código-fonte** em 8 linguagens (Java, Kotlin, Python, Go, JS/TS, C#, Ruby, PHP), com evidência `arquivo:linha`. Imagem base `scratch` (~3 MB), **100% offline** e determinística. Deliberadamente conservador: só emite criptografia que reconhece — **nunca marca nada como "seguro"** (o julgamento é do motor).

Construir a imagem. O que este passo faz: cross-compila o binário estático no host e o copia numa base `scratch` — nada é baixado.

```
make codescan-image
```

Você deve ver:

```
  cross-compilando lastrium-codescan (linux/<arch>)…
  imagem: lastrium-codescan:test  (sha256:…)
  produção: publique num registry e fixe scanner.image por <ref>@sha256:<digest> (o motor avisa quando não-pinado)
```

Usar no `ci`. O que este passo faz: o motor roda o container com `--network=none` (padrão seguro), monta seu repositório em `/src:ro` e consome o CBOM do stdout.

```
./lastrium ci --source ./repo --scanner-image lastrium-codescan:test --offline
```

Você deve ver: no stderr do scanner, `lastrium-codescan: N componente(s) criptografico(s) em /src`, e os artefatos do `ci` em `./lastrium-out`.

Usar direto (contrato cru), sem o `ci`:

```
docker run --rm -v "$PWD/repo":/src:ro lastrium-codescan:test dir /src > cbom.json
```

5.4 `lastrium-deepscan` — o scanner profundo (Java, com rede)

Análise **semântica** de Java via SonarQube 26.x embarcado + plugin **sonar-cryptography 1.6.0** (IBM/PQCA). O ganho sobre o leve: resolve o uso real das APIs — `Cipher.getInstance("AES/CBC/PKCS5Padding")` vira `AES-128-CBC-PKCS5`, `RSA` vira `RSA-2048`, assinaturas viram `RSA-PKCS1-1.5-SHA-256`.

Limitações importantes (por design da base SonarQube):

- **Pesado**: imagem ~1 GB; cada scan sobe um SonarQube embarcado → **~2–3 min**.
- **Precisa do namespace de rede**: o Elasticsearch do SonarQube falha com `--network=none` (a análise em si é loopback-only, sem chamadas externas — mas use **apenas com esta imagem first-party**).
- **Só Java** (hoje); para as demais linguagens, o codescan cobre.
- O plugin 1.6.0 **só carrega em SonarQube 26.x** (9.9/10.x falham com "Fail to load plugin").

Construir a imagem. O que este passo faz: baixa o plugin (URL fixada no `Makefile`: `https://github.com/cbomkit/sonar-cryptography/releases/download/1.6.0/sonar-cryptography-plugin-1.6.0.jar`) e faz o `docker build` de `deploy/deepscan-sonar/` (requer REDE).

```
make deepscan-image
```

Você deve ver: `imagem: lastrium-deepscan:test (~1000 MB)`.

Usar no `ci`. O que este passo faz: igual ao codescan, mas liberando a rede do container e alargando o timeout (o default de 120 s não basta para o boot do SonarQube).

```
./lastrium ci --source ./repo \
  --scanner-image lastrium-deepscan:test \
  --scanner-allow-network \
  --scanner-timeout 300
```

> O `Makefile` recomenda `--scanner-timeout 300`; o README do deepscan usa `360`. Em máquinas lentas, prefira 360.

Usar direto (contrato cru):

```
docker run --rm -v "$PWD/repo":/src:ro lastrium-deepscan:test dir /src > cbom.json
```

Você deve ver: no stderr, o progresso `[deepscan] iniciando SonarQube embarcado (aguarde ~60-90s)…` → `[deepscan] SonarQube UP` → e o CBOM no stdout.

5.5 `cbomkit-theia` — o scanner de filesystem (PQCA/IBM)

Scanner aberto da PQCA/IBM para **infraestrutura**: certificados X.509, chaves PEM, keystores, configs TLS/SSH. É o default do contrato do `ci` (os `--scanner-args` padrão são `dir,/src`, exatamente o entrypoint do theia). O laboratório (seção 10) o usa com a tag local `cbomkit-theia:test`; a obtenção/ build da imagem é feita a partir do projeto upstream da PQCA **[confirmar o procedimento de build na sua infraestrutura — o repositório do motor não embute a receita]**.

```
./lastrium ci --source ./infra --scanner-image cbomkit-theia:test --offline
```

5.6 "Traga seu CBOM" (air-gap total)

Qualquer scanner que emita CycloneDX 1.6 serve. Coloque o arquivo em `<source>/cbom.cdx.json` (ou aponte `--source caminho/arquivo.json`) e rode o `ci` **sem** `--scanner-image` — nenhum container é executado.

5.7 Validação detalhada

1. **Contrato do scanner**: rode o container direto (`docker run … > cbom.json`) e confira que a saída é JSON CycloneDX: `python3 -c "import json;d=json.load(open('cbom.json'));print(d['bomFormat'],d['specVersion'],len(d.get('components',[])))"` → **você deve ver** `CycloneDX 1.6 <N>`.
2. **Exit code do container** = 0; diagnósticos vão para o stderr, **nunca** misturados ao CBOM do stdout.
3. **Offline de verdade (codescan)**: o `ci` roda o scanner com `--network=none` por padrão — não passe `--scanner-allow-network` e observe que o scan completa mesmo sem internet.
4. **Evidência rastreável**: no CBOM do codescan, cada componente carrega a evidência `arquivo:linha` do uso detectado.
5. **Teste negativo (deepscan)**: rode o deepscan **sem** `--scanner-allow-network` — o SonarQube não sobe (`UnknownHostException` do Elasticsearch) e o `ci` termina com erro operacional. É o comportamento esperado, não um defeito.

5.8 Problemas comuns

- **Timeout no deepscan** — o default é 120 s e o SonarQube leva ~60–90 s só para subir. Use `--scanner-timeout 300` (ou 360).
- **Imagem não-pinada** — em produção, publique a imagem num registry e fixe `scanner.image` por `<ref>@sha256:<digest>` no `.lastrium.yml`; o motor avisa quando a referência não está pinada.
- `docker` **ausente** — use `--scanner-runtime podman`, ou caia no modo "traga seu CBOM" (5.6), que não precisa de container.
- **Codescan não achou nada** — ele varre extensões de fonte conhecidas e pula diretórios de dependência (`node_modules`, `vendor`, `target`, `build`, `dist`, …). Arquivos acima de 2 MiB são ignorados (heurística de fonte).

---

6. `lastrium attest` — predicate CBOM para cosign

6.1 O que é e quando usar

Lê um SBOM/CBOM CycloneDX, roda a análise de exposição quântica e emite um **predicate in-toto** com a postura criptográfica do artefato — o *dogfooding* do pipeline (cada release do Lastrium pode carregar o CBOM avaliado pelo próprio Lastrium). Use quando quiser **anexar a postura criptográfica a uma imagem/artefato** via `cosign attest`, no ecossistema de supply-chain que sua organização já usa.

6.2 Os dois predicates do produto (são DISTINTOS, por design)

PREDICATETYPE	EMITIDO POR	DESCREVE	SCHEMA PUBLICADO
`https://lastrium.com.br/cbom/v1`	`lastrium attest --sbom`	a **avaliação de exposição** de UM artefato (tool/source/summary/assets) — vira atestação portátil via `cosign attest`	`docs/attestation/cbom-predicate-v1.schema.json`
`https://lastrium.com.br/attestations/cryptoposture/v1`	`lastrium ci`	a **execução do pipeline** de CI (engine/scanner/pack/gate/totais), no envelope DSSE `attestation.dsse.json` que sela o dossiê	`docs/attestation/cryptoposture-predicate-v1.schema.json`

Quando usar qual: `cbom/v1` é a atestação portátil de um artefato/imagem; `cryptoposture/v1` é a prova da execução do `ci`, verificável offline com `lastrium verify` (o núcleo do dossiê). Política de versão: mudança **aditiva opcional** fica na mesma URL (schema atualizado); mudança **breaking** vira `/v2` com URL e schema novos, publicados ao lado — nunca sobrescrevendo (`docs/attestation/VERSIONING.md`).

6.3 Flags (todas — confirmadas no código)

FLAG	SIGNIFICADO
`--sbom <path>`	caminho do SBOM/CBOM CycloneDX (**obrigatório**)
`--out <path>`	arquivo de saída (padrão: stdout)
`--in-toto`	emitir o **Statement in-toto completo** (subject + predicateType + predicate) em vez de só o predicate

Exit codes: `0` ok; `2` erro operacional (sem `--sbom`, arquivo ilegível etc.).

6.4 Uso passo a passo

1. **Gerar o predicate.** O que este passo faz: avalia o SBOM (mesmo caminho do `assess`) e serializa o predicate.

```
./lastrium attest --sbom sbom.cdx.json --out cbom-predicate.json
```

Você deve ver (forma real, com `testdata/sample-cbom.json`):

```
{
  "tool": { "name": "lastrium", "version": "0.3.3" },
  "source": { "bomFormat": "CycloneDX", "specVersion": "1.6" },
  "summary": {
    "totalCryptoAssets": 9, "vulnerable": 2, "weakened": 5, "postQuantum": 1,
    "notApplicable": 0, "indeterminate": 1, "critical": 2,
    "uniqueVulnerableAlgorithms": 2
  },
  "assets": [ ... ]
}
```

`generatedAt` **só aparece** com `SOURCE_DATE_EPOCH` definido (determinismo: o attest nunca usa `time.Now`).

2. **Atestar a imagem com cosign.** O que este passo faz: o cosign monta o Statement com `subject = <image>@<digest>` e assina — usa-se o predicate **solto** (sem `--in-toto`).

```
cosign attest --yes \
  --type https://lastrium.com.br/cbom/v1 \
  --predicate cbom-predicate.json \
  "$CI_REGISTRY_IMAGE@sha256:..."
```

3. **Qualquer um verifica:**

```
cosign verify-attestation \
  --type https://lastrium.com.br/cbom/v1 \
  "$CI_REGISTRY_IMAGE:vX.Y.Z" ...
```

4. **Statement completo (fluxos sem cosign).** O que este passo faz: emite o Statement in-toto inteiro, com o subject sendo o **próprio CBOM analisado** (`name` = nome do arquivo, `digest.sha256` = hash dos bytes — honesto: atesta-se sobre ESTE documento).

```
./lastrium attest --sbom sbom.cdx.json --in-toto
```

Você deve ver: `"_type": "https://in-toto.io/Statement/v1"`, `"predicateType": "https://lastrium.com.br/cbom/v1"` e o `subject` com o sha256 do arquivo.

5. **Interop com o envelope do `ci` (o outro predicate).** O envelope `attestation.dsse.json` do `ci` também é verificável por ferramentas cosign:

```
cosign verify-blob-attestation \
  --key attest.pub \
  --type https://lastrium.com.br/attestations/cryptoposture/v1 \
  --signature out/attestation.dsse.json \
  <artefato>
```

> A verificação primária e offline continua sendo `lastrium verify` (stdlib puro); o cosign é o caminho de interop para quem já o usa.

6.5 Validação detalhada

1. **Exit codes**: `0` com `--sbom` válido; `2` sem `--sbom` (mensagem `lastrium attest: --sbom é obrigatório` no stderr) ou com arquivo ilegível.
2. **Conformidade ao schema publicado**: o predicate deve validar contra `docs/attestation/cbom-predicate-v1.schema.json` (e o do `ci` contra `cryptoposture-predicate-v1.schema.json`). O consumidor roteia o schema pela URL do `predicateType` — nunca assume um schema fixo.
3. **Determinismo**: `SOURCE_DATE_EPOCH=1750000000 ./lastrium attest --sbom X` duas vezes → bytes idênticos (mesmo `generatedAt`); sem a variável, `generatedAt` é omitido.
4. **Verificação cosign**: o `cosign verify-attestation --type https://lastrium.com.br/cbom/v1 <image>` deve passar após o `cosign attest`.
5. **Privacidade**: o predicate publicado contém apenas algoritmo, status, severidade, ameaça, localização e nome do componente — **nunca** valores de segredos nem o caminho absoluto da máquina (o subject usa o *basename*). Atenção: `location` / `source.component` são passthrough do CBOM de entrada — garanta que seu scanner emita **caminhos relativos** antes de publicar em registry público.

6.6 Problemas comuns

- **`generatedAt` sumiu** — comportamento correto sem `SOURCE_DATE_EPOCH`; defina a variável se quiser o timestamp reproduzível.
- **Subject "errado" no cosign** — no fluxo com `cosign attest`, o subject vem do argumento de imagem; o predicate deve ir **solto** (sem `--in-toto`). O modo `--in-toto` é para fluxos que montam o DSSE manualmente.
- **Caminhos absolutos vazando no predicate** — vêm do scanner que gerou o CBOM (passthrough fiel); ajuste o scanner para caminhos relativos ao projeto.

---

7. `lastrium dossier` + `lastrium verify` — a prova para o auditor

7.1 O que é e quando usar

`dossier` empacota os artefatos de um run do `ci` num **arquivo único, autossuficiente e determinístico** (tar com `manifest.json` + LEIA-ME). `verify` confere, **offline e num comando**, a **autoria** (assinatura Ed25519 da atestação) e a **integridade** (SHA-256 de cada artefato). É o que o auditor/BCB executa — sem depender do Lastrium nem de internet.

7.2 Pré-requisitos

- Um diretório `out/` gerado pelo `lastrium ci` **com** `--attest-key` (seção 4). O dossiê aceita atestação sem assinatura, mas avisa que não prova autoria.
- A **chave pública** (`attest.pub`) — é tudo de que o auditor precisa.

7.3 Flags (todas — confirmadas no código)

`lastrium dossier`:

FLAG	SIGNIFICADO
`--in <dir>`	diretório com os artefatos do `lastrium ci` (**obrigatório**)
`--out <path>`	caminho do dossiê a gerar (**obrigatório**), ex.: `dossie.lastrium`

Exit: `0` ok; `2` erro operacional (dir vazio, sem `attestation.dsse.json` — mensagem manda rodar `lastrium ci` antes).

`lastrium verify` :

FLAG	SIGNIFICADO
`--key <pub.pem>`	chave pública Ed25519 PEM (**obrigatório**)
`--artifacts-dir <dir>`	ao verificar um envelope DSSE **solto**: confere a integridade dos artefatos no diretório
`--rekor-url <url>`	**[reservado]** aceito por compatibilidade, ignorado com aviso

O argumento posicional é o arquivo a verificar — **tanto** um `attestation.dsse.json` solto **quanto** um dossiê (o `verify` detecta sozinho). As duas formas funcionam: `verify <arquivo> --key k` e `verify --key k <arquivo>`.

Exit codes: `0` assinatura válida e (se aplicável) todos os digests batem; `1` assinatura inválida, digest divergente, artefato ausente, statement sem subjects ou envelope não assinado; `2` erro operacional (arquivo ausente, PEM inválido, flag inválida).

7.4 Uso passo a passo

1. **Verificar o envelope + artefatos do `ci`.** O que este passo faz: valida a assinatura Ed25519 do envelope DSSE e recalcula o SHA-256 de cada artefato do diretório, comparando com os *subjects* assinados.

```
./lastrium verify out/attestation.dsse.json --key attest.pub --artifacts-dir out/
echo "exit=$?"
```

Você deve ver (saída real):

```
lastrium verify: assinatura Ed25519 OK
lastrium verify: OK cbom.cdx.json (e7e6e7a0861c406a...)
lastrium verify: OK conformidade-bcb538.json (f4ea6c4c1bf351ea...)
lastrium verify: OK custodia.json (b996ed30d8385a7f...)
lastrium verify: OK gap-report.json (2f894b50bfbad144...)
lastrium verify: OK gl-sast-report.json (3c82a7b1dc0fee06...)
lastrium verify: OK lastrium.sarif (56c34cd4dd53926f...)
lastrium verify: OK report.md (cd573f8ab5e85cf7...)
lastrium verify: OK summary.json (88bcc5fe263e05fd...)
lastrium verify: todos os artefatos íntegros
exit=0
```

2. **Empacotar o dossiê.** O que este passo faz: junta todos os artefatos num tar determinístico com índice legível. Para timestamp reproduzível, use `SOURCE_DATE_EPOCH` (sem ela, o campo "Gerado em" sai `1970-01-01T00:00:00Z` — época Unix, política de determinismo, não um bug).

```
./lastrium dossier --in out --out dossie.lastrium
```

Você deve ver (saída real):

```
lastrium dossier: dossie.lastrium (9 artefatos, 282624 bytes)
  assinado por keyid 284150490e734f116ecc21538a4d71357aa0614e6e4746ad3e6c245a9f62811b
  verifique com: lastrium verify dossie.lastrium --key <chave-publica.pem>
```

(Se a atestação não estiver assinada, sai o aviso `AVISO: atestação NÃO assinada — o dossiê não prova autoria (defina LASTRIUM_ATTEST_KEY no "lastrium ci")`.)

3. **O auditor verifica o dossiê — OFFLINE, um comando.** O que este passo faz: abre o tar, valida a assinatura da atestação embutida, recalcula o SHA-256 de cada artefato **embutido** e imprime o veredito de auditoria.

```
./lastrium verify dossie.lastrium --key attest.pub
```

Você deve ver (saída real): as linhas `OK …` por artefato e o bloco final

```
===== DOSSIÊ DE CONFORMIDADE — VÁLIDO E ÍNTEGRO =====
Gerado em:   1970-01-01T00:00:00Z
Motor:       0.3.3   Pacote regulatório: bacen-br@2026.06-go
Signatário:  keyid 284150490e734f116ecc21538a4d71357aa0614e6e4746ad3e6c245a9f62811b
Ativos:      52   Vulneráveis: 21 (7 algoritmos distintos)
Migração:    P1=0 P2=0 P3=0 P4=0   incompletos=21
Base norm.:  CMN 5.274 / BCB 538 — art. 3º, §2º, II [a confirmar] · LGPD — art. 46 [a confirmar] · ...
=====================================================
```

com **exit 0**. O resumo de conteúdo vem do `summary.json` — que é coberto pela atestação (logo, confiável após a verificação); o `manifest.json` é só índice.

7.5 Validação detalhada — incluindo o teste negativo didático

1. **Caminho feliz**: passos 1 e 3 acima → exit 0, `todos os artefatos íntegros`.
2. **TESTE NEGATIVO (o mais importante do produto): adultere 1 byte e veja o `verify` apontar o arquivo.** O que este passo faz: acrescenta um único espaço ao final de um artefato — simulando uma edição maliciosa da evidência — e verifica de novo.

```
cp out/summary.json out/summary.json.bak     # backup
printf ' ' >> out/summary.json               # adultera 1 byte
./lastrium verify out/attestation.dsse.json --key attest.pub --artifacts-dir out/
echo "exit=$?"
```

Você deve ver (saída real): os demais artefatos `OK`, e

```
lastrium verify: DIGEST DIVERGENTE "summary.json"
  esperado: 88bcc5fe263e05fdf0310116ce9d929c5d1b7199bee354c2796ec138c0462cff
  obtido:   ab07b40f625f05be7f3ef868ebd43f01f8106e80b202222b3d4585d8e99a1c1c
lastrium verify: FALHA — adulteração detectada em um ou mais artefatos
exit=1
```

O `verify` reporta **todas** as divergências (não para na primeira) e o exit é **1**. Isso vale igualmente para um documento de evidência de governança (`evidencia--…`) ou para o `conformidade-bcb538.json` alterados.

3. **Restaure e confirme a volta ao verde:**

```
mv out/summary.json.bak out/summary.json
./lastrium verify out/attestation.dsse.json --key attest.pub --artifacts-dir out/
echo "exit=$?"    # deve voltar a 0
```

4. **Chave errada** → `lastrium verify: ASSINATURA INVÁLIDA — …`, exit 1.
5. **Envelope sem assinatura** → `ERRO — atestação não assinada (signatures vazio)`, exit 1 (não é sucesso silencioso).
6. **Artefato removido do diretório** → `ARTEFATO AUSENTE "<nome>"`, exit 1.

7. **Diferencie os exits**: `1` = "isto foi adulterado/é inválido" (fraude); `2` = "não consegui verificar" (arquivo/chave ausente, PEM inválido) — operacional, não veredito.

7.6 Problemas comuns

- **`"attestation.dsse.json" não encontrado em <dir>`** (exit 2 do `dossier`) — o diretório `--in` não é a saída de um `lastrium ci`; rode o `ci` antes.
- **"Gerado em: 1970-01-01T00:00:00Z"** — esperado sem `SOURCE_DATE_EPOCH` (timestamp determinístico); defina a variável no empacotamento se quiser a data real do release.
- **Verificou o envelope solto mas esqueceu `--artifacts-dir`** — sem a flag, só a assinatura é conferida (exit 0 não cobre integridade dos arquivos ao lado). Para integridade completa, use `--artifacts-dir out/` ou o dossiê.

---

8. `lastrium revocation` — OCSP/CRL (opt-in)

8.1 O que é e quando usar

Verificação de **revogação de certificados** (OCSP, com CRL) como **evidência assinada pela CA** — separada do `ci` / `assess` para preservar o offline-first: **por padrão não acessa a rede**. Use quando precisar provar, com evidência de terceiro, que os certificados do inventário não estavam revogados na data da consulta — e sele essa evidência num dossiê.

8.2 Pré-requisitos

- CLI instalada; um diretório com os **certificados** (públicos) a verificar.
- Para o modo `--online`: acesso de rede aos endpoints OCSP/CRL das CAs.

8.3 Flags (todas — confirmadas no código)

FLAG	DEFAULT	SIGNIFICADO
`--root <dir>`	`.`	diretório raiz com os certificados a verificar
`--online`	`false`	consultar OCSP/CRL na rede (**opt-in**; padrão OFFLINE)
`--out <dir>`	— (stdout)	diretório de saída: `revocation-evidence.json` + respostas assinadas da CA
`--as-of YYYY-MM-DD`	—	data da consulta registrada na evidência
`--timeout <s>`	`8`	timeout por requisição, em segundos

Exit codes: `1` se **algum certificado está REVOGADO** (sinal para o CI); `2` erro operacional; `0` caso contrário.

8.4 Uso passo a passo

1. **Modo offline (inventário de endpoints).** O que este passo faz: varre os certificados e registra os endpoints OCSP/CRL encontrados — **sem** tocar a rede. O status sai `indeterminado` (nunca "bom": sem consultar a CA, não se afirma nada).

```
./lastrium revocation --root ./certs
```

Você deve ver: um JSON no stdout com `"online": false` e os itens encontrados (num diretório sem certificados: `{"online": false, "itens": null}`), exit 0.

2. **Modo online (evidência assinada).** O que este passo faz: consulta OCSP (POST) com fallback CRL (GET), grava `revocation-evidence.json` e as **respostas assinadas da CA** em `--out` — evidência de terceiro, pronta para ser selada num dossiê (basta o `--out` apontar para o diretório que o `dossier` empacota).

```
./lastrium revocation --root ./certs --online --out ./out --as-of 2026-07-04
```

Você deve ver (stderr):

```
lastrium revocation: N certificado(s), M evidência(s) assinada(s) em ./out
```

e, em `./out/`, o `revocation-evidence.json` + os blobs de resposta da CA.

8.5 Validação detalhada

1. **Offline nunca conclui "bom"**: sem `--online`, todo status é `indeterminado` — se você vir um "bom" sem rede, é bug; reporte.
2. **Evidência de terceiro**: as respostas OCSP/CRL gravadas em `--out` são **assinadas pela autoridade certificadora** — verificáveis independentemente do Lastrium.
3. **Exit code como gate**: num CI, `lastrium revocation --online … ; test $? -eq 1` detecta certificado revogado (exit 1) e permite reprovar a esteira.
4. **Determinismo do registro**: `--as-of` fixa a data registrada na evidência (o motor não usa o relógio para o conteúdo avaliativo).
5. **Selagem**: rode o `revocation --out ./out` **antes** do `lastrium ci … --out ./out`? Não — o `ci` gera a atestação apenas sobre o que ele próprio emite. Para selar a evidência de revogação, inclua-a no diretório e empacote com `lastrium dossier --in ./out` (o dossiê embute todos os arquivos do diretório; a *prova de autoria* continua sendo a atestação do `ci`, e a evidência OCSP/CRL carrega a assinatura da própria CA).

8.6 Problemas comuns

- **Timeout nas consultas** — o default é 8 s por requisição; aumente com `--timeout` em redes lentas/proxies.
- `HTTP 4xx/5xx` **do endpoint** — a CA não respondeu OK; o item fica sem resposta conclusiva → `indeterminado` (nunca "bom").
- **Esperava que o** `ci` **fizesse isso sozinho** — não faz, por design: o `ci` é offline; a revogação é etapa separada e opt-in.

---

9. Portal (web) — Área do Cliente e verificador de selo

9.1 O que é e quando usar

O portal de licenciamento em **licensing.lastrium.com.br** tem duas caras: a **Área do Cliente** (cadastro, downloads do app e da CLI, ativação/gestão de licença por HWID) e o **verificador público de selo** (`/verificar`). Use a primeira para obter o software; a segunda, sempre que alguém precisar conferir a autenticidade de um laudo.

9.2 Pré-requisitos

Navegador. Para a Área do Cliente, uma conta (e-mail/senha).

9.3 Uso passo a passo — Área do Cliente

1. Acesse **https://lastrium.com.br/area** e clique em **Cadastrar** (e-mail, senha, nome). **Você deve ver:** o e-mail de boas-vindas com o link de download.
2. Baixe o que precisar: - App desktop: `Lastrium_<versão>_aarch64.dmg` (macOS), `.deb` / `.rpm` (Linux). - CLI: `lastrium-<versão>-<os>-<arch>` (binário único; `chmod +x`).
3. Licença do app: informe o **HWID** mostrado pelo app no campo próprio; ao receber a chave, ative no app (seção 1.3).

9.4 Uso passo a passo — Verificar um selo

1. Acesse **https://licensing.lastrium.com.br/verificar**.
2. Escaneie o QR do laudo (ou cole o conteúdo lido) e clique **Verificar**. O que este passo faz: o portal decifra o conteúdo, valida a assinatura do selo e mostra o veredito.
3. **Você deve ver: "✓ Selo AUTÊNTICO e ÍNTEGRO"** com emissor, data e hash — que deve bater com o SHA-256 impresso no laudo.

9.5 Validação detalhada

1. **Veredito positivo**: selo de um laudo legítimo → "AUTÊNTICO e ÍNTEGRO", hash idêntico ao do PDF.
2. **Teste negativo**: conteúdo truncado/adulterado do QR → o selo **não** valida.
3. **Privacidade**: quem intercepta o QR sem o portal vê apenas texto cifrado; ao selar, **só o hash** do laudo trafega ao portal — nunca o conteúdo.
4. **Postura de segurança do portal** (declarada): MFA/TOTP, CSRF por sessão, rate-limit, sessões revogáveis, senhas argon2id.

9.6 Problemas comuns

- **Selo "não validado"** — conteúdo do QR incompleto, ou selo emitido por outro ambiente/chave.
- **Não achou o download da sua plataforma** — os artefatos publicados são macOS Apple Silicon (`aarch64.dmg`), Linux x86_64 (`.deb` / `.rpm`) e os binários da CLI por SO/arquitetura; para outras combinações, compile a CLI do fonte (seção 2.4).

---

10. Laboratório Banco Meridiano — a demonstração completa

10.1 O que é e quando usar

Um **banco fictício completo** (core banking + PIX com TLS real, JWT RS256, ledger SHA-256, comprovante Ed25519) rodando em Docker, sobre o qual o Lastrium executa o fluxo inteiro: **inventariar → avaliar → mapear à norma → selar a prova**. Use para demonstração a CISO/Risco/auditoria (~8 min) ou para estudar o produto de ponta a ponta com dados realistas. Tudo é gerado localmente; nenhum dado é real. Fica em `lab/` no repositório do motor.

10.2 Pré-requisitos

- **Docker + docker compose**.
- A imagem do scanner **`cbomkit-theia:test`** disponível localmente (confira: `docker images | grep theia`). Sem ela o passo de scan de infra falha — dá para rodar só com o inventário de código (ver 10.6).
- `bash`, `python3` e `openssl` no host.
- Opcional (para **selar** o laudo): um token de licença em `pipeline/laudo/.lab-license` (uma linha) ou `export LICENSE_TOKEN=…`. Sem token, o laudo sai como pré-visualização **não selada** (com hash, sem QR).

Conferência rápida:

```
docker version >/dev/null && echo "docker ok"
docker images | grep -q cbomkit-theia && echo "scanner ok" || echo "FALTA a imagem cbomkit-theia:test"
python3 --version && openssl version && make --version | head -1
```

10.3 Uso passo a passo — o caminho rápido

```
cd lab
make all      # scan → avaliação → relatório → laudo → banco vivo → web
```

Você deve ver, ao final (a 1ª execução constrói as imagens; o motor com OPA/Rego leva ~2 min):

```
═══════════════════════════════════════════════════
 LABORATÓRIO PRONTO — Banco Meridiano
   • Relatório (web):  http://localhost:8091
   • Relatório (PDF):  out/relatorio-meridiano.pdf  (executivo/técnico)
   • Laudo selado:     out/laudo-meridiano.pdf  (QR → licensing.lastrium.com.br/verificar)
   • Dossiê de prova:  out/dossie-meridiano.lastrium  (lastrium verify)
   • Banco vivo:       core TLS em https://localhost:8443/healthz
═══════════════════════════════════════════════════
```

10.4 Cada alvo do `make`, explicado

ALVO	O QUE FAZ	O QUE VOCÊ DEVE VER
`make bank`	sobe o `meridiano-core` (HTTPS/TLS) e roda o job `meridiano-pix`: transferência Alice → Bob assinada como **JWT RS256**, liquidada com comprovante **Ed25519** e ledger **SHA-256**	o log do PIX com o token `eyJ...`, `LIQUIDADO ✓`, o comprovante `TXN000001` e os saldos finais. Confira: `curl -sk https://localhost:8443/healthz` → `ok`
`make scan`	gera o **estate** (PKI: certificados, chaves, keystores) se preciso, roda o **cbomkit-theia** sobre a infra → `scan/cbom-estate.json`, gera o inventário de código → `scan/code-inventory.cbom.json` e mescla → `scan/cbom-completo.json`	as contagens dos três arquivos (na versão atual do tutorial: 68 infra + 22 código = 90 componentes; versões anteriores do lab citam 73 — o total varia com a versão do estate)
`make assess`	constrói a imagem `meridiano-lastrium` (motor **com OPA/Rego**) e roda `lastrium ci` sobre o inventário: artefatos + atestação + **dossiê**, verificado ali mesmo	a lista de artefatos, o `gate exit` e o bloco `DOSSIÊ DE CONFORMIDADE — VÁLIDO E ÍNTEGRO`
`make relatorio`	gera o **Relatório de Postura Criptográfica** completo (HTML + PDF, executivo/técnico) via Node + Chromium em container	`relatório (PDF): /out/relatorio-meridiano.pdf`
`make laudo`	monta o laudo jurídico, calcula o SHA-256, pede o **selo** ao portal (se houver licença) e embute o QR	`impressão digital (SHA-256): …`, `selo: selado · <seal_id>`, `laudo (PDF): /out/laudo-meridiano.pdf`
`make web`	sobe o painel nginx	**http://localhost:8091**
`make down`	encerra os containers (preserva `out/`)	—
`make clean`	encerra e apaga tudo que foi gerado (estate, CBOMs, `out/`)	`limpo.`

10.5 Onde cada peça de prova aparece

ONDE	O QUE VER
http://localhost:8091	o painel da demo: KPIs, distribuição por status, prioridades de migração (P1–P4), tabela de mapeamento regulatório filtrável, seção de evidência assinada — e a barra **"Gerar documentos"** com os botões **Gerar Relatório (PDF)** e **Gerar Laudo selado (PDF)** (equivalem a `make relatorio` / `make laudo`, sem terminal)
`out/relatorio-meridiano.pdf`	o relatório completo (executivo/técnico)
`out/laudo-meridiano.pdf`	o laudo **selado** (hash + QR → verificar no portal)
`out/report.md`	o relatório técnico cru por ativo
`out/gap-report.json`	o mapeamento norma↔ativo bruto
`out/dossie-meridiano.lastrium`	o dossiê verificável offline

10.6 Validação detalhada

1. **Banco vivo**: `curl -sk https://localhost:8443/healthz` → `ok`.
2. **Inventário real**: `python3 -c "import json;print(len(json.load(open('scan/cbom-completo.json'))['components']),'ativos')"` → o total de componentes do merge.
3. **Dossiê íntegro (a validação-chave, offline):**

```
lastrium verify out/dossie-meridiano.lastrium --key pipeline/attest.pub
# ou, sem CLI no host:
docker run --rm -v "$PWD":/lab -w /lab meridiano-lastrium \
  verify out/dossie-meridiano.lastrium --key pipeline/attest.pub
```

Você deve ver: `todos os artefatos íntegros` + o bloco `DOSSIÊ DE CONFORMIDADE — VÁLIDO E ÍNTEGRO`, exit 0.

4. **Selo do laudo**: abra `out/laudo-meridiano.pdf`, vá à página de autenticidade e verifique o QR em **https://licensing.lastrium.com.br/verificar** → **"Selo AUTÊNTICO"**.
5. **Teste negativo**: aplique o teste de adulteração da seção 7.5 sobre os artefatos de `out/` — o `verify` tem de apontar o arquivo alterado, exit 1.
6. **O mesmo inventário no app**: arraste `scan/cbom-completo.json` para o app desktop — é o caminho do usuário final sobre os mesmos dados.

10.7 Problemas comuns

- **`port is already allocated` na 8091** — troque a porta no `docker-compose.yml` (serviço `meridiano-web`, `ports: ["8091:80"]` → `8092:80`).
- **Falta a imagem `cbomkit-theia:test`** — o scan de infra falha. Opções: (a) obter/gerar a imagem do cbomkit-theia (projeto PQCA/IBM); (b) rodar só com o inventário de código, usando `scan/code-inventory.cbom.json` como `scan/cbom-completo.json`.
- **Laudo saiu "não selado"** — faltou a licença (10.2); adicione o token e rode `make laudo` de novo (o portal precisa estar acessível). O token de demo do lab é dedicado (`Laboratorio Lastrium DEMO (NAO PRODUCAO)`) — não distribua; para não selar contra produção, `export SEAL_API=http://localhost:9999`.
- **`make assess` lento na 1ª vez** — é o build do motor com OPA/Rego (~2 min); as execuções seguintes usam o cache do Docker.
- **Permissões em `out/`** — o Makefile roda os containers com o seu usuário (`--user $(UID):$(GID)`); mantenha isso se editar o Makefile.

---

11. Validação de ponta a ponta em 10 minutos

O roteiro mínimo que prova o produto inteiro funcionando — do self-test à detecção de fraude. Pré-requisitos: CLI instalada (seção 2), `openssl`, e o repositório do motor (pelos CBOMs de exemplo em `testdata/`; sem o repositório, baixe https://lastrium.com.br/exemplos/cbom-meridiano-grande.json).

```
# [1] O motor está são? (self-test das invariantes do classificador)
./lastrium healthcheck
#    → "ok lastrium 0.3.3", exit 0

# [2] Avaliação básica funciona? (e o "indeterminado nunca é seguro"?)
./lastrium assess testdata/sample-cbom.json
#    → relatório com 9 ativos; "Obfusca-XOR" sai indeterminado, com o aviso
#      "NAO classificados como seguros"; exit 0

# [3] Chaves de atestação (uma vez)
openssl genpkey -algorithm ed25519 -out attest.key
openssl pkey -in attest.key -pubout -out attest.pub

# [4] Pipeline completo, assinado e determinístico
mkdir -p src && cp testdata/cbom-meridiano-grande.json src/cbom.cdx.json
SOURCE_DATE_EPOCH=$(date +%s) ./lastrium ci --source src --out out \
  --attest-key attest.key --offline
ls -1 out/
#    → exit 0; 9 artefatos, incluindo attestation.dsse.json e conformidade-bcb538.json

# [5] Verificação: assinatura + integridade de TODOS os artefatos
./lastrium verify out/attestation.dsse.json --key attest.pub --artifacts-dir out/
#    → "assinatura Ed25519 OK", uma linha "OK <artefato>" por arquivo,
#      "todos os artefatos íntegros", exit 0

# [6] Dossiê: a prova num arquivo único
./lastrium dossier --in out --out dossie.lastrium
./lastrium verify dossie.lastrium --key attest.pub
#    → "DOSSIÊ DE CONFORMIDADE — VÁLIDO E ÍNTEGRO", exit 0

# [7] TESTE NEGATIVO: adultere 1 byte → o verify TEM de falhar apontando o arquivo
cp out/summary.json out/summary.json.bak
printf ' ' >> out/summary.json
./lastrium verify out/attestation.dsse.json --key attest.pub --artifacts-dir out/
#    → 'DIGEST DIVERGENTE "summary.json"' + hashes esperado/obtido +
#      "FALHA — adulteração detectada", exit 1

# [8] Restaure → volta ao verde
mv out/summary.json.bak out/summary.json
./lastrium verify out/attestation.dsse.json --key attest.pub --artifacts-dir out/
#    → "todos os artefatos íntegros", exit 0

# [9] (Bônus, +1 min) Determinismo byte-a-byte
SOURCE_DATE_EPOCH=1750000000 ./lastrium ci --source src --out outA --attest-key attest.key --offline
SOURCE_DATE_EPOCH=1750000000 ./lastrium ci --source src --out outB --attest-key attest.key --offline
shasum -a 256 outA/attestation.dsse.json outB/attestation.dsse.json
#    → os dois hashes idênticos
```

Se os passos [1]–[8] se comportaram exatamente como descrito, você provou: o classificador (com a regra "indeterminado nunca é seguro"), o pipeline de artefatos, a atestação Ed25519, o dossiê, a verificação offline **e** a detecção de adulteração — o produto inteiro, sem internet.

Apêndice A — Tabela-resumo de exit codes

COMANDO	0	1	2	3
`healthcheck`	saudável	degradado	—	—
`assess`	avaliado	erro	—	—
`ci`	ok / warn	violação (modo fail)	erro operacional	licença inválida
`attest`	ok	—	erro operacional	—
`dossier`	ok	—	erro operacional	—
`verify`	válido e íntegro	inválido / adulterado / não assinado	erro operacional	—
`revocation`	ok	certificado REVOGADO	erro operacional	—
`license verify`	válida	expirada / inválida	erro operacional	—

Apêndice B — Lacunas encontradas nas fontes (marcadas no texto)

- **`.AppImage` na Área do Cliente** — a pipeline gera o artefato (`desktop/README.md` , `.gitlab-ci.yml`), mas o material publicado documenta apenas `.deb` / `.rpm` como caminho de download. **[confirmar]**
- **Obtenção da imagem `cbomkit-theia:test`** — o laboratório a exige local, mas o repositório não embute a receita de build (projeto upstream PQCA/IBM). **[confirmar]**
- **Contagem de ativos do laboratório** — o tutorial (`lab/docs/01`) espera 90 componentes (68+22); o `lab/README.md` cita 73. O total varia com a versão do estate; use a contagem impressa pelo seu `make scan` .
- **Timeout do deepscan** — o `Makefile` sugere `--scanner-timeout 300` ; o README do deepscan, `360` . Ambos funcionam; em máquinas lentas prefira 360.
- **URLs diretas de download dos instaladores** — ficam atrás do login da Área do Cliente (https://lastrium.com.br/area); não há URL pública direta nas fontes.

Dúvidas: suporte@lastrium.com.br · lastrium.com.br · © Lastrium — evidência de postura, não certificação legal.