VISOR WORKSHOP

Open-source SDLC automation and code review orchestration.

PRESENTING THIS DECK

```
npm run workshop:setup   # one time; pins reveal-md
npm run workshop:serve   # starts local server (watch mode)
# Exports
npm run workshop:export  # static HTML → workshop/build
npm run workshop:pdf     # PDF → workshop/Visor-Workshop.pdf
```

AGENDA (ICEBERG FORMAT)

- Surface: What Visor is and quick start
- Layer 1: Core concepts and defaults
- Layer 2: Code review pipeline (overview → security → performance → quality → style)
- Layer 3: Customizing (tags, dependencies, templates, prompts)
- Layer 4: Architecture & internals
- Layer 5: SDLC automations (cron, webhooks, HTTP, Jira, release notes)
- Layer 6: Nested runs, foreach/loops
- Layer 7: Debugging, logging, observability
- Layer 8: Extending providers and advanced recipes

WHAT IS VISOR?

Config-first automation for code review and SDLC workflows with native GitHub checks/annotations.

- Runs locally as a CLI and in CI/GitHub Actions
- Produces structured, predictable outputs (JSON, Markdown, SARIF)
- Composable checks with dependencies, tags, and templates
- Multi-provider AI (or no-AI) and HTTP/command integrations

90-SECOND QUICK START

Action (minimal):

```yaml
# .github/workflows/visor.yml
name: Visor
on:
  pull_request: { types: [opened, synchronize] }
  issues: { types: [opened] }
  issue_comment: { types: [created] }
permissions:
  contents: read
  pull-requests: write
  issues: write
  checks: write
jobs:
  visor:
    runs-on: ubuntu-latest
    steps:
```

CLI (in this repo):

```
npx -y @probelabs/visor --output table
```

LAB 0 — FIRST RUN (2 MIN)

1. Run defaults locally (all checks):

```
npx -y @probelabs/visor --output table --debug
```

2. Try JSON output to a file:

```
npx -y @probelabs/visor --check security --output json --outpu
```

3. Filter by tags (fast/local):

```
npx -y @probelabs/visor --tags local,fast --max-parallelism 5
```

CORE CONCEPTS

- Check: unit of work (e.g., `security`)
- Schema: JSON shape for outputs (e.g., `code-review`)
- Template: how results are rendered
- Group: comment bucket (`overview`, `review`, etc.)
- Provider: execution engine (`ai`, `http`, `command`, `claude-code`, `log`)
- Dependencies: `depends_on` defines order; independents run in parallel
- Tags: label checks (`fast`, `local`, `comprehensive`) and filter via `--tags`
- Events: PRs, issues, comments, webhooks, or cron

THE DEFAULT PIPELINE

`overview` → `security` → `performance` → `quality` → `style` with session reuse and GitHub annotations.

```
# defaults/.visor.yaml (excerpt)
checks:
  overview:    { type: ai, group: overview }
  security:    { type: ai, group: review, depends_on: [overview
  performance:{ type: ai, group: review, depends_on: [security
  quality:     { type: ai, group: review, depends_on: [performa
  style:       { type: ai, group: review, depends_on: [quality]
```

LAB 1 — USING DEFAULTS (3 MIN)

Run only the `overview` and `security` checks:

```
npx -y @probelabs/visor --check overview,security --output tab
```

Add `--debug` to see dependency decisions and timing.

CODE REVIEW WORKFLOW

Visor emits native GitHub check runs and inline annotations.

- Schemas ensure predictable, renderable outputs
- Comments grouped for easy scanning
- Suppress false positives with `// visor-disable`

PR COMMENT COMMANDS

Trigger from comments:

```
/review
/review --check security
/visor how does caching work?
```

LAB 2 — SUPPRESSIONS (2 MIN)

Add a suppression near a flagged line:

```
const testPassword = "demo123"; // visor-disable
```

Re-run and confirm the warning is suppressed.

CUSTOMIZING VISOR

Start from defaults, extend for your repo’s needs.

TAGS AND PROFILES

```yaml
checks:
  security-quick:
    type: ai
    prompt: "Quick security scan"
    tags: [local, fast, security]
```

CLI:

```
npx -y @probelabs/visor --tags local,fast
```

DEPENDENCIES AND ORCHESTRATION

```
checks:
  security:    { type: ai }
  performance: { type: ai, depends_on: [security] }
```

Independent checks run in parallel; dependent checks observe order.

TEMPLATES AND PROMPTS

Place prompts in files and render via Liquid:

```
checks:
  overview:
    type: ai
    prompt: ./prompts/overview.liquid
```

Render JSON in debug templates with `| json`.

LAB 3 — YOUR FIRST CONFIG (5 MIN)

Open `workshop/labs/lab-01-basic.yaml` and run:

```
npx -y @probelabs/visor --config workshop/labs/lab-01-basic.ya
```

Tweak a prompt and rerun. Then add a tag and filter by it.

ARCHITECTURE

High-level flow:

COMPONENTS (MENTAL MODEL)

- CLI and Action entrypoints (Node 18+)
- Config manager (load/merge/extends)
- Orchestrator (graph, parallelism, retries, fail-fast)
- Providers: `ai`, `command`, `http`, `http_client`, `log`, `claude-code`
- Renderers: JSON → templates → outputs

SDLC AUTOMATIONS

Examples beyond PR review:

- Release notes (manual, tag-driven)
- Cron audits against main
- HTTP/webhooks (receive → run checks → respond)
- Jira workflows and status sync

DEMO TARGETS (FROM EXAMPLES/)

- `examples/cron-webhook-config.yaml`
- `examples/http-integration-config.yaml`
- `examples/jira-simple-example.yaml`
- `defaults/.visor.yaml` (release notes)

Run one locally:

```
npx -y @probelabs/visor --config examples/http-integration-con
```

LAB 4 — RELEASE NOTES (5 MIN)

Simulate a release notes generation:

```
TAG_NAME=v1.0.0 GIT_LOG="$(git log --oneline -n 20)" \
GIT_DIFF_STAT="$(git diff --stat HEAD~20..HEAD)" \
npx -y @probelabs/visor --config defaults/.visor.yaml --check :
```

NESTED RUNS AND LOOPS

Use `forEach`/loop patterns for multi-target checks.

See: `examples/forEach-example.yaml`, `examples/for-loop-example.yaml`.

LAB 5 — FOREACH (5 MIN)

Run the foreach example and observe dependency propagation:

```
npx -y @probelabs/visor --config examples/forEach-example.yaml
```

DEBUGGING & OBSERVABILITY

- `--debug` for verbose execution tracing
- Use `log()` inside `if:` and `transform_js:` expressions
- Dump context in templates via Liquid `| json`
- Emit `json` or `sarif` and save with `--output-file`

LAB 6 — DEBUG & LOGS (3 MIN)

Run the debug example config:

```
npx -y @probelabs/visor --config workshop/labs/lab-03-debug.ya
```

Open the log output and correlate with the rendered markdown.

PROVIDERS AND EXTENSIBILITY

Mix and match providers:

- `ai` — model-based checks (Gemini/Claude/OpenAI/Bedrock)
- `command` — run shell tasks; great for linters/tests
- `http`/`http_client` — call external APIs or receive webhooks
- `log` — structured logging to outputs
- `claude-code` — deeper analysis via Claude Code SDK

MINIMAL COMMAND PROVIDER EXAMPLE

```yaml
checks:
  unit-tests:
    type: command
    exec: 'npm test --silent'
    on: [manual]
```

Run:

```
npx -y @probelabs/visor --config workshop/labs/lab-02-command.y
```

LAB 4 — CLASSIFY → SELECT → PLAN (8–10 MIN)

End-to-end planner that:

- Classifies a task description into components and checks
- Runs per-component agents (with folder-scoped context)
- Consolidates into a single implementation proposal

Run it fast with the mock provider:

```
npx -y @probelabs/visor --config workshop/labs/lab-04-planner.y
```

Optional: provide your own task via env var

```
TASK_DESC="Add caching to HTTP client without breaking retries
npx -y @probelabs/visor --config workshop/labs/lab-04-planner.
```

BUILDING YOUR OWN PROVIDER (CONCEPTUAL)

1. Implement a provider module (inputs → run() → outputs)
2. Register it in config, choose a schema/template
3. Reuse orchestration (tags, deps, templates) for free

See: `docs/pluggable.md` and `docs/command-provider.md`.

CHEATSHEET

CLI sampling:

```
# Run all checks from current config (defaults if none)
npx -y @probelabs/visor --output table

# Filter by tags
npx -y @probelabs/visor --tags local,fast

# JSON/SARIF outputs
npx -y @probelabs/visor --check security --output json --outpu
npx -y @probelabs/visor --check security --output sarif --outp

# Use a specific config
npx -y @probelabs/visor --config workshop/labs/lab-01-basic.ya

# Debugging
npx -y @probelabs/visor --debug
```