ChatGPT

Design Constraint Validator Documentation Audit (Phase C.1)

Documentation Landscape Summary

What exists: The Design Constraint Validator (DCV) repository has a **rich set of documentation**, including a detailed root README, a dedicated `/docs` directory with multiple guides (Getting Started, Constraint types, CLI reference, Configuration, Architecture, API, etc.), some **defensive publication** documents under `docs/prior-art/` (which capture design intent and research context), and various **example READMEs** demonstrating usage with real token setups (Style Dictionary, Tokens Studio, etc.). There are also developer-focused docs like CONTRIBUTING and RELEASE guides. In addition, the docs in `/docs` are duplicated as a GitHub wiki (per the provided Wiki setup instructions), though the wiki content is meant to mirror the `/docs` folder [1] [2].

How it evolved: Earlier in the project, documentation was more ad-hoc and scattered. The root README and a standalone **Configuration Guide** were likely the initial sources of truth for users. Over time (likely around the v1.0 release in late 2025), the maintainers undertook a comprehensive documentation effort, consolidating content into a structured `/docs` folder for a more cohesive experience. This evolution is evident from overlapping content: for example, there are two separate configuration docs (one in root, one in `/docs`), and the root README contains usage info that also appears in the new guides. The presence of "prior art" design docs dated **November 7, 2025** [3] [4] suggests that around the time of first stable release, the author formalized the architecture and conceptual underpinnings. In summary, DCV's documentation has **grown from a single README into a multi-file manual**, plus research write-ups – a sign of a maturing project.

Cohesiveness vs. fragmentation: The current documentation set is **comprehensive but somewhat fragmented**. On the positive side, the `/docs` content is well-organized and targeted to different audiences (beginners, regular users, developers) [5], creating a potentially cohesive "official manual." The defensive publications add valuable architectural rationale but live in a separate corner (`prior-art`) not directly referenced in user-facing docs. Meanwhile, duplication issues (e.g. two configuration guides, overlapping content between README and docs/Home) make it unclear which source is most up-to-date. Overall, the documentation **landscape is extensive and informative**, but would benefit from pruning of outdated duplicates and clearer pointers to the canonical info.

Documentation Map Table

Document	Purpose / Summary	Apparent Age	Status (Guess)
README.md (root)	Introduction, value proposition of DCV, badges, quick install and usage examples (validate, why, graph), and explanation of DCV's philosophy ("constraints, not conventions") [6]. Links to an AI Guide, examples, and shows basic outputs.	Evolved over project; content suggests mid-to-late development (updated with latest features like cross-axis)	**Canonical-ish but overlapping** – Still a primary entry point, but some info duplicates newer docs. Generally up-to-date, except minor flag mismatches (e.g. uses `--format json` whereas CLI doc uses `--summary`) – indicates slight drift.
CONFIGURATION.md (root)	Standalone Configuration Guide – how DCV finds token files, themes (constraints), and overrides by default; how to customize via `dcv.config.*` or package.json [7] [8]. Provides examples of basic and full config JSON, explains all config options (tokens, themes, overrides, breakpoints, output paths, validation settings) [9] [10], and covers file discovery details and expected token/ constraint file formats [11] [12].	Likely **early-mid** (written when custom config was introduced). Some terminology (e.g. uses `"never"` instead of `"off"` for `failOn` value) suggests it predates final naming [13] [14].	**Outdated/duplicative** – Much content overlaps with the newer docs/Configuration.md. Has minor inconsistencies (e.g. `failOn: "never"` vs `"off"`) indicating it wasn't fully updated [13] [14]. Should be merged or deprecated in favor of the docs/ version.
CONTRIBUTING.md (root)	Guidelines for contributors: how to set up the dev environment, run tests, code style, commit message conventions, PR process, and release steps [15] [16].	Probably maintained continuously; no obvious version tied – content mentions npm scripts for release (consistent with 1.0 release process).	**Current (developer-facing)** – Seems up-to-date for maintaining the project (references to release scripts align with RELEASE.md). Not user documentation, but no red flags here.

Document	Purpose / Summary	Apparent Age	Status (Guess)
RELEASE.md (root)	Maintainer guide for publishing to npm: login, version bump, tagging, publishing, verifying, and using the automated GitHub Actions workflow [17] [18].	Written around initial releases (refers to v1.0.0 and semantic versioning) – likely **recent** with v1.0 launch.	**Current (maintainer-facing)** – Specific to internal release workflow. Doesn't impact end-users, and appears accurate for now.
docs/README.md ("Documentation" index)	Serves as an index for docs, with Quick Links to all major guides [19]. Explains which docs are for beginners vs. users vs. developers [5]. Also provides links to the main README, config guide, contributing, examples, and instructs how to mirror docs to the GitHub Wiki [20].	Created **recently** (as part of the docs revamp, presumably for v1.0). Integrates all pieces in one structure.	**Canonical** – Central hub for documentation. Should remain the primary nav for users. (One quirk: it links to the *root* CONFIGURATION.md as "Configuration Guide", implying dual sources for config info).
docs/Home.md (Wiki Home)	The landing page content for the GitHub wiki, largely mirroring the main README and docs index. Includes an intro blurb, a list of documentation sections with brief descriptions, quick links for installation, basic commands, and resources [21] [22]. It reiterates "What is DCV" and the philosophy [23] [6].	**Recent** (part of docs reorganization to support wiki). Essentially a curated copy of content from root README and docs index, tailored for wiki.	**Duplicate** – Content overlaps heavily with README.md (e.g. the "Why constraints" section appears in both [6] [24]). Maintained as part of wiki sync, but introduces redundancy.

Document	Purpose / Summary	Apparent Age	Status (Guess)
docs/Getting-Started.md	Step-by-step beginner's tutorial. Covers installation (local, global, npx) [25], prerequisites (Node 18+, design token basics) [26], then **Your First Validation** in 3 steps: creating a sample `tokens.json`, writing basic constraint files (`themes/wcag.json` for contrast, `themes/typography.order.json` for hierarchy) [27] [28], and running `dcv validate`. Shows expected pass output [29], then guides the user to introduce an error (making h2 larger than h1) to see a violation report [30] and explains the error breakdown [31]. Finishes with next steps: exploring examples, other commands (`why`, `graph`, `build`), customizing configuration (with an example `dcv.config.json`) [32], adding more constraints with a summary of all five types [33], common issues (file not found, no constraints, CI failing builds) with solutions [34] [35], and how to get help.	**Recent** – Likely written for the formal docs launch. Incorporates features up to cross-axis constraints, so it's up-to-date.	**Canonical** – The go-to quickstart for new users. Very hands-on and appears consistent with current code behavior. (No obvious outdated info; the examples align with actual example files. It even references the latest command flags like `--fail-on off` for CI use [36].)

docs/Constraints.md ("Constraint Types")	Detailed guide to all supported constraint types. For each of Monotonic, WCAG contrast, Threshold, Lightness, and Cross-Axis, it describes use cases and configuration format. E.g., Monotonic: used for typographic scales, spacing, breakpoints; configured via `"order": [...]` arrays in a `*.order.json` file [37]. WCAG: explains AA/AAA contrast standards, shows a `wcag.json` example with multiple rules [38] [39] and a table of WCAG ratios [40]. Threshold: ensures min/max (e.g. ≥44px touch targets); interestingly, the doc shows threshold rules as TypeScript snippet (implying they might be built-in) [41] rather than a JSON schema example. Lightness: ordering of color shades by perceptual lightness (uses OKLCH); example `color.order.json` fragment given [42] [43]. Cross-Axis: the most complex – enforces conditional multi-property rules (IF X THEN Y); doc provides multiple real-world examples (readability for light fonts, touch target on mobile, responsive contrast, heading emphasis, dense spacing) with JSON examples of `rules` objects [44] [45]. Also defines the syntax of a rule (with `when` and `require` clauses and available helper functions) [46] [47], and how to scope rules per breakpoint by file naming	**Recent** – Reflects the full feature set including the latest "⚡ Cross-Axis" constraints, which likely were added towards the end of development. The presence of emoji and detailed examples suggests it was written once the feature set stabilized (probably right before 1.0).	**Canonical (with a caveat)** – This is the authoritative reference for constraint syntax. The only issue is a slight **ambiguity around Threshold constraints**: unlike other types, the doc doesn't show how to declare them in JSON (it only gives a code snippet [41]), which could confuse users on how to use threshold rules. It hints these may be predefined (the snippet lists a built-in rule id `control.size.min`), but doesn't explicitly say if users can add their own. This part of the doc might be incomplete or assumes the built-ins cover it. Otherwise, the content is up-to-date and very informative.

Document	Purpose / Summary	Apparent Age	Status (Guess)
	(`.sm.rules.json`, etc.) [48] [49].		

docs/CLI.md (“CLI Reference”)	Exhaustive list of all CLI commands and options. It starts with global options (tokens path, config path, verbosity, breakpoint selection) [50], then documents each command in turn: `validate` (with options like `--fail-on`, `--summary` (table vs json output), `--strict`, performance budgets, etc. [51] [52]), including examples and exit codes [53]; `graph` (options to choose format mermaid/dot/json, filters, Hasse diagram generation, violation highlighting) [54] [55] plus usage examples; `why` (to explain token provenance); `build` (to generate token outputs in CSS/JS/JSON, with options like `--format`, `--output`, `--watch`, and the `--mapper` to apply a CSS var mapping [56]); `set` (to set a token value and produce a patch) [57]; `patch` and `patch:apply` (exporting differences and applying them to tokens) [58]. The CLI doc provides example outputs (e.g., sample table and JSON from `validate` [59] [60], and graph format examples in Mermaid, DOT, JSON [61] [62]).	**Recent-ish, but possibly slightly lagging** – It appears to cover new commands like `set` and `patch`, which suggests it was updated when those features were added (likely late in development). However, some parts of the JSON output examples don’t match the final schema (for instance, CLI.md’s JSON example uses keys `severity` and `kind` [63], whereas the actual output schema uses `ruleId` and `level` [64]). It also documents a `--summary json` flag for `validate` [65], whereas elsewhere (README, AI Guide) the usage `--format json` is shown [66] – indicating a rename that not all docs reflected.	**Mostly Canonical, minor staleness** – It’s the primary reference for CLI usage. The presence of **inconsistent flag naming and JSON field names** is a red flag (likely an artifact of evolving the CLI interface). For example, if `--summary` was changed to `--format` in final implementation (or vice versa), the docs should uniformly reflect one. Currently the main README uses `--format` (e.g., `npx dcv validate --format json`) [66] while CLI.md uses `--summary json` [67]. Similarly, the JSON output schema in CLI.md vs JSON-OUTPUT.md differ. These discrepancies suggest the CLI doc needs a refresh to be fully canonical.

Document	Purpose / Summary	Apparent Age	Status (Guess)
docs/ Configuration.md ("Configuration" guide)	A thorough guide to customizing DCV. It overlaps with the root CONFIGURATION.md but is more detailed and updated. Explains how DCV discovers config (order of precedence: dcv.config.json, .dcvrc, package.json, etc.) [68] and the zero-config defaults (expects `tokens.json` and `themes/` folder by default, etc.) [69]. Then it provides the config JSON schema: first a "Basic Configuration" example [70], then a "Full Configuration" example covering all fields (tokens, themes, overrides, breakpoints, output, validation, graph, constraints) [71] [72]. After that, each configuration option is explained in its own subsection (paths for tokens/ themes/overrides [73] [74], breakpoints array [75], output paths [76], validation settings like failOn/quiet/strict/ summary [77] [78], graph settings [79] [80], and finally a note that constraints can be defined directly in the config JSON as well). It also likely covers advanced usage (multiple token files via glob or array, etc., as hinted by examples) and maybe environment variable overrides (the Home page mentioned environment variables, but need to confirm if this doc covers them).	**Recent** – Written as part of the new docs set. It uses the current terminology (`"off"` instead of "never" for failOn) [14] and includes new options (e.g., `summary` output format, graph settings) that indicate it's aligned with the latest features.	**Canonical** – This should replace the root CONFIGURATION.md as the authoritative config reference. It appears up-to-date with correct option names and defaults. One possible omission: it doesn't explicitly mention environment variables in the snippet we saw (the Home.md mentioned "Environment variables" as a topic for config [81], but the actual content in this file wasn't visible in our excerpt – worth verifying if present). Overall, aside from that check, this doc is comprehensive and current.

docs/ Architecture.md	An **internal architecture overview** of DCV's engine. Walks through the internal pipeline in phases: **Phase 1: Token Parsing** (adapters normalize various token formats into a flat map [82] [83], then references are resolved so that dependencies are explicit [84] [85]), **Phase 2: Dependency Graph** (build a directed acyclic graph of token dependencies [86], representing tokens as nodes and reference or constraint relationships as edges [87] [88]; includes cycle detection logic) [89] [90], **Phase 3: Constraint Validation** (plugin architecture – each constraint type is implemented as a plugin conforming to a `ConstraintPlugin` interface; example code for MonotonicPlugin and WcagContrastPlugin is given [91] [92], showing how they fetch token values via an Engine API and push Violation objects on failures [93] [94]). It also describes the Engine's API (get/set tokens, register plugins, run validation, get the graph) [95] [96]. Subsequent sections (Phase 4: Provenance Tracing) explain how the `why` command works (traversing the graph to find origins of a token's value) [97], multi-breakpoint support (merging base tokens with override files for each breakpoint and validating each set) [98], and **Performance optimizations** (incremental validation and	**Recent** – This document reads like a capstone overview likely written or finalized around the 1.0 release. It references all key features (including breakpoints, cross-axis, etc.) and aligns with the defensive publications (e.g., it emphasizes post-compute validation and engine-agnostic design, which echo the prior-art doc themes).	**Canonical (for internal design)** – This is the definitive guide to how DCV works under the hood. It's detailed and up-to-date. One thing to watch: it mirrors some content in the prior-art papers (like the focus on "post-compute" and receipts), but with slight differences in emphasis. There's no obvious contradiction, just parallel explanations. As long as the code continues to match these descriptions (and it seems to), this doc is in good shape.

Document	Purpose / Summary	Apparent Age	Status (Guess)
	parallel subgraph validation) [99] . It also covers **Color Handling** details (color space conversion, contrast calculation, alpha compositing) [100] [101] , error handling philosophy (validation errors vs runtime errors) [102] , and extension points like how to add **Custom Plugins** and **Custom Adapters** (with stub code examples for each) [103] . Finally, a note on testing strategy.		

Document	Purpose / Summary	Apparent Age	Status (Guess)
docs/API.md (“API Reference”)	Documentation for using DCV programmatically via its Node.js/TypeScript API. It shows how to import the package (ESM only) [104], and gives examples for each main function: `validate(options)` – with the `ValidateOptions` interface (paths for tokens, themes, etc. similar to CLI flags) and the `ValidationResult` format [105] [106], plus usage example code [107]. Similarly `build(options)` with its result shape [108] [109], `graph(options)` [110] [111], and `why(token, options)` with `ProvenanceResult` structure [112] [113]. It also covers the lower-level **Engine class** (constructor, methods) for advanced use or writing custom plugins [114] [115], and likely mentions any utility exports.	**Recent** – Created along with the other docs. It references features like breakpoints and highlights that the package is ESM-only (a relatively new Node 18+ aspect). The types and function options match what’s in CLI, so it’s concurrent with v1.0.	**Canonical** – Should be accurate as the source of truth for developers embedding DCV. No obvious issues; it aligns with the CLI and config docs. (One minor note: it doesn’t explicitly define some terms like what keys appear in a Violation object, but the types make it fairly clear. Also, if the JSON output schema changed, we should ensure it’s reflected here – e.g., whether `Violation.kind` vs `ruleId` is used. The API doc uses `Violation.kind` in the interface [116], whereas JSON-OUTPUT.md uses `ruleId` [117]. This might be a naming discrepancy between internal type vs output serialization. We should confirm and unify naming in docs to avoid confusion.)

Document	Purpose / Summary	Apparent Age	Status (Guess)
docs/JSON-OUTPUT.md ("JSON Output Schema")	Focused documentation on the machine-readable JSON outputs and receipts. It describes how to use `--format json` and `--output` to get JSON results [118], then defines the **ValidationResult** schema in detail [119] [120] (slight variation in property names from API doc: uses `violations` & `warnings` arrays, `counts` object with `checked/violations/warnings` counts, and `stats` with duration, version, timestamp [117] [121]). It then defines the **ConstraintViolation** structure [64] with fields like `ruleId`, `level` (error/warn), `message`, `nodes`, `edges`, `context` for extra info [117]. An example JSON output is given for clarity [122] [123]. The doc also enumerates the CLI **Exit Codes** (0=success, 1=violations, 2=config error, 3=runtime error) and how `--fail-on` affects them [124] [125] – this overlaps with CLI.md but is good to have here too. It covers **Receipt Mode**: using `--receipt` to generate an audit JSON, and defines the **ValidationReceipt** schema (basically ValidationResult plus `environment`, `inputs`, and `config` sections) [126] [127] with an example [128] [129]. Finally, it provides integration tips (like a GitHub Actions snippet uploading the JSON) [130].	**Recent** – Almost certainly written at 1.0 time, given receipts and JSON outputs were a key "polished" feature. The content is very detailed and matches the latest design (notice it uses `ruleId` and `level` terminology, which likely reflects final naming).	**Canonical** – This doc should be the single source of truth on output formats. It's precise and up-to-date. The only concern is consistency: other docs and the code should use the same terminology (for instance, ensuring that the CLI and API docs refer to these fields identically – earlier, CLI.md used `kind`/`severity` in an example, which should be revised to `ruleId`/`level` as per this schema [63] [117]). As long as that alignment is done, JSON-OUTPUT.md stands as a reliable reference.

Document	Purpose / Summary	Apparent Age	Status (Guess)
docs/AI-GUIDE.md (“AI Assistant Guide”)	A somewhat meta document – aimed at helping ChatGPT/ Copilot or other AI assist DCV users. It provides **copy-paste ready** examples of common commands (validate, why, graph, build) for quick suggestion [131] [132]. It lists typical user questions and direct answers with the appropriate DCV command (e.g. “How do I validate my tokens?” -> `npx dcv validate tokens.json` [133]; “I want to see errors but not fail build” -> use `--fail-on off` [134]; “Can I get machine-readable output?” -> yes, `--format json --output …` [135]; “visualize relationships?” -> `dcv graph --format mermaid` [136]). It summarizes the JSON output format (in a brief schema snippet) [137] [138] and shows the expected project file structure (tokens.json, themes/*.json, etc.) [139] to orient helpers. It also gives quick examples of each constraint type for AI to reference when explaining (monotonic, WCAG, threshold JSON snippets) [140] [141]. Finally, it covers programmatic API usage in brief, some troubleshooting Q&A, installation variations, and common integration patterns (CI snippet, pre-commit hook example).	**Recent** – This is clearly added around the docs overhaul, anticipating users with AI assistance. It references current features and uses current flags.	**Non-user-Facing (current)** – This guide is up-to-date and unique in purpose. It’s not exactly part of the user manual, but it doesn’t conflict with it either. Status is fine, just that it might not need to be exposed to end-users directly (it’s more for AI/ maintainer reference).

Document	Purpose / Summary	Apparent Age	Status (Guess)
docs/WIKI-SETUP.md	Maintainer note on how to copy the docs into the GitHub wiki. It lists which pages should exist in the wiki (Home, Getting Started, Constraints, CLI, Configuration, Architecture – and notes the API page is missing and should be added) [142] [143]. It then gives step-by-step instructions to create the wiki pages and a sidebar with the documentation links [1] [144]. Emphasizes keeping the docs in sync between `/docs` and wiki, and benefits of each format [145] [146].	**Recent** – Written when the decision was made to offer documentation in both the repository and the wiki. Likely around the time of docs launch (no earlier need for it).	**Ancillary** – Only relevant to maintainers. It's complete and not likely to go stale unless the doc set changes (e.g., they already noted the API page needed adding to wiki). No issues – might be removed or moved out of user docs directory to avoid clutter, but not harmful.

docs/prior-art/DCV-Post-Compute-Validation-and-Receipts.md	A defensive publication (licensed CC BY 4.0) establishing prior art for DCV's core concepts [147] . It reads like an academic or patent disclosure: includes Abstract, Motivation, Method, etc. Key points: **Post-compute validation** – DCV runs *after* design tokens are fully computed into an **EffectiveConfig** (flattened map of what actually ships) [148] . It uses **Policy Packs** (collections of constraints: WCAG, monotonic, thresholds, etc.) [148] [149] to validate and emits a machine-readable **receipt** with hashes of inputs/policies and results [148] . The document justifies this approach by pointing out shortcomings of conventional token linters (they often run *before* final values are known, lack composability, and produce only human-readable reports) [150] , whereas DCV's method addresses those by being engine-agnostic, post-compute, plugin-based, and producing a durable JSON receipt [151] . It defines Input/Output **contracts** (EffectiveConfig schema example, Policy pack contents, optional dependency graph metadata) [152] [153] , describes the validation pipeline with a diagram [154] , details the need for determinism & hashing for reproducibility [155] [156] , and enumerates the supported **Constraint Kinds** (Monotonic, WCAG, Threshold, Lightness, Cross-	**Dated Nov 7, 2025** (just before/around initial release) [3] , but conceptual. It represents the **design philosophy and intended architecture** at that time.	**Valid but Parallel** – This is not user documentation but an architecture reference. It aligns with the implemented features (we see those exact concepts in DCV). However, it's somewhat frozen in time as a snapshot of design intent. As long as DCV stays on this design path, the document isn't "outdated" per se; but if new constraints or major changes occur, this paper wouldn't reflect them (by design, since it's prior art). No contradictions noticed – if anything, it's more detailed about rationale than the user docs. It should be considered a **reference design document** rather than a living doc to update with each release.

Document	Purpose / Summary	Apparent Age	Status (Guess)
	Axis) with brief descriptions [157] [158] – basically the theoretical counterpart to what's in docs/Constraints. It then defines the JSON result/receipt schemas similarly to JSON-OUTPUT.md (though using slightly different naming – e.g. it calls `counts` as part of result, etc.) and notes about breakpoints and provenance.		

docs/prior-art/ Decision-Themes-Deterministic-Compute-and-Dual-Namespaces.md	Another defensive publication (same date, CC BY 4.0) covering a broader theming engine concept (“Decision Themes”) which DCV can validate. It describes computing a design system from two inputs: a **Visual Theme (VT)** of literal design primitives (colors in OKLCH, base sizes, etc.) and a **Decision Theme (DT)** of relations (ratios, mappings, constraints) [159] [160]. The method yields a deterministic **EffectiveConfig** (flattened tokens with a “golden hash”) and then runs post-compute validation (via DCV) to produce a receipt [161] [162]. It introduces terms: **Dual-Namespace** (using separate CSS variable prefixes for the editor UI vs the exported design system – `--ui-*` vs `--ds-*`) to avoid circular dependencies in a design tool [161] [163], and insists on an explicit “Apply” action to push changes to external systems, generating a receipt for audit [161] [164]. Essentially, this document is painting the context in which DCV operates: it’s engine-agnostic, so any design tool that follows these contracts (VT+DT producing an EC) can use DCV to validate. It provides example data structures for VT, DT, EC (with dual-namespaced tokens) [160] [165], explains the motivation (problems with traditional token systems and how this approach solves them) [166] [167], and outlines the method	**Dated Nov 7, 2025** [4]. This represents **early design thinking** that likely preceded or paralleled DCV development. It’s more “forward-looking” about how a design tool could be built in tandem with DCV.	**Contextual / Possibly Out of Scope** – It’s relevant for understanding terms like “EffectiveConfig” or why the DCV design emphasizes engine agnosticism and receipts. However, for DCV’s own documentation, this is somewhat tangential. There’s no evidence that DCV’s codebase implements anything about dual-namespacing or VT/DT computation – those are upstream concerns. So this doc stands as background context. It doesn’t conflict with DCV’s docs, but average DCV users might find it confusing or irrelevant to their use of the validator alone. It’s accurate as an idea paper, but not a part of the main product documentation.

Document	Purpose / Summary	Apparent Age	Status (Guess)
	step-by-step (compute, hash, validate, receipt, apply) [168].		

(Note: There are also various example READMEs under `/examples/` *(for tokens from Tokens Studio, Style Dictionary, advanced constraints, etc.), which serve as usage demonstrations. They aren't core design docs so they're not individually listed above, but they do provide additional guidance in context. For instance, the* `examples/dtcg/README.md` *likely shows how to use DCV with the W3C Design Tokens Community Group format, etc.)*

Terminology Map

Design Token – A granular design value, typically defined in a JSON with a `$value`. In DCV, tokens are the input data being validated. They are often nested and referenced by dot-notated **token IDs** after flattening (e.g. `typography.size.h1`). DCV can ingest tokens in various formats via **adapters** (Style Dictionary format, DTCG, etc.), normalizing them to a common shape [82] [83]. No major naming issues here: "token" is used consistently to mean a design system variable like a color, size, etc.

Constraint – A rule or requirement that design tokens must satisfy. This is the core concept of DCV (hence the name). Constraints enforce relationships (not just static checks) – e.g., "token X ≥ token Y" or "contrast ratio ≥ 4.5:1". In config files, different constraint types are indicated either by top-level keys like `"order": [...]` for monotonic, or nested under `"constraints": { "wcag": [ … ] }` for contrast, etc. [37] [169]. **Synonyms/related terms:** In prose, sometimes referred to as "rules" (especially in context of policy packs or cross-axis "rules"). The prior-art docs call a collection of constraints a **"Policy Pack"** [149]. Importantly, DCV's docs make a point that these are *not* mere lint checks but mathematical constraints. The term is used consistently throughout, though in code one sees "plugins" implementing constraints. No conflicting usage, just that a newbie might see "policy" or "theme" (see below) and not immediately realize it refers to sets of constraints.

Violation – An instance where a token (or tokens) violates a constraint. DCV outputs **violations** after validation. In JSON output, each violation has a type (ruleId/kind), severity (error or warning), a message, and context like involved token nodes or edges [117] [170]. The term "violation" is consistently used in results and docs to denote a failure of a constraint (e.g., "3 violations found"). No alternate term is used for this concept, and its meaning is clear. One thing to note: older docs or code used the term "issues" or "errors" sometimes interchangeably, but "violation" is now the standard term. For example, the API result uses `result.violations` (and counts of errors/warnings separately) [171].

Policy / Policy Pack – This term appears mainly in the defensive publication and a bit in README. It refers to a **set of constraints**, often corresponding to a standard or an organization's rules. For instance, an "AA policy" might include WCAG AA contrast rules, etc. The README usage: "Validate with a policy profile (e.g., AA)" [172] suggests a single JSON that aggregates certain constraints. In the CLI, there was a `--policy` flag example to point to a policy file. However, the official docs don't elaborate on "policy" much; instead they speak in terms of constraint files in a themes directory. So, **there is potential confusion**: *Policy Pack* is essentially the same as a set of constraint JSON files. The code's config uses `themes` key to denote where constraints live, not a `policy` key (though the CLI example shows `--policy` usage). In short, "policy" is more a conceptual grouping (and used in writings) whereas the implementation uses "themes" directory or individual constraint files. These should be clarified in docs: e.g., if `--policy` is supported as a shortcut to load one file or set of constraints, it needs documentation. Right now, *Policy* and *Constraints* are used in similar contexts, which could be unified.

Theme – This is a loaded term with two distinct meanings in the documentation: 1. In the **design tokens world**, a "theme" is often a collection of tokens (like a light theme vs dark theme). 2. In DCV's config, `themes` is ironically the directory for constraint files by default [173]. The use of "themes" here likely stemmed from considering each set of constraints as a "theme" or from the notion of design themes that include constraints.

Furthermore, the prior-art **Decision Themes** doc uses "Visual Theme" and "Decision Theme" to mean something entirely different (input data vs relational rules) [160]. This could be very confusing if one doesn't realize that DCV's `themes/` folder has nothing to do with Visual/Decision themes, but is just named that way. There's an **inconsistency** in usage: - In user docs, when they say "themes directory", they mean "constraints directory". They sometimes even phrase it as "Path(s) to constraint/theme files" [173]. - The Decision Themes paper's concepts are not referenced in user docs, so there's not a direct clash for end-users, but internally the term "theme" had broader meaning.

Manifest – Surprisingly, this term is *not prominent in DCV's docs*. It might refer to design token "manifest" in a generic sense or perhaps to a mapping file for CSS variables. We saw the CLI tests mention a `--mapper examples/manifest.example.json` which likely maps token names to CSS variable names [174]. However, in documentation, there isn't a section explicitly about a "manifest". Possibly the Style Dictionary or DTCG examples use a manifest concept, or it's the CSS var mapping file (mapper) referred to in CLI. Since the user specifically inquired about "manifest," we infer it's about mapping design tokens to final CSS var names (some design token tools use a manifest file). DCV's CLI does support a `--mapper` for build [56], documented as "CSS variable mapping manifest". So, **Manifest** in DCV context likely means a JSON file that maps canonical token names to legacy names or different naming schemes for output. It's a minor feature, and the term isn't used elsewhere. There's no conflict, just that it's not widely explained outside the CLI reference.

Poset – Short for *partially ordered set*. DCV's monotonic constraints essentially define a poset of tokens (e.g., a hierarchy of sizes). The term appears in the README ("Hasse/poset graph export") [175], the CLI (the `--hasse` option generates a Hasse diagram of a poset [176]), and architecture doc (which references building a DAG for dependencies – a poset is related but not identical concept, though monotonic constraints form a poset graph). The code even has a `core/poset.ts`. In documentation aimed at users, "poset" is used sparingly and not deeply defined, likely assuming an advanced user would know it means the simplified graph of ordering relations. They do explain visually via Hasse diagram outputs rather than define the term formally. There is no inconsistent usage (it always refers to the ordering relationships). However, it might be jargon for some readers – perhaps why they often pair it with "Hasse diagram" (which is a more explanatory way to show the poset).

EffectiveConfig – This term comes from the prior-art docs, referring to the fully **computed** token set after merging all inputs (the "effective" configuration that actually goes into a product). It's capitalized in those documents and treated as an important concept (the thing you validate *against*). In user-facing docs, this term is not really used (because DCV operates on whatever token JSON you give it – which presumably is already the effective set). But understanding it helps clarify DCV's post-compute stance. No conflicts, just that it's a concept mostly in the design papers.

Receipt – A JSON report that DCV can output, containing the validation results plus hashes and environment info for audit. The term is used in both the user docs (JSON-OUTPUT.md has a section on "Receipt Mode (Audit Trail)" [177]) and the prior-art where it's central. It's consistently used to mean the artifact file (not to be confused with just the results printed on console). No synonyms; usage is clear and consistent.

Breakpoint – In DCV, breakpoints refer to responsive design sizes (like sm, md, lg) for which tokens might have override values. The documentation covers this: default breakpoints are ["sm","md","lg"] [178], and DCV can validate each by merging override token files. The term is used exactly as in CSS context. No conflicting usage; just to note that cross-axis rules can even incorporate breakpoints (some cross-axis examples check if `bp === 'sm'` for mobile rules [179]).

Overrides – This refers to token override files for different breakpoints. The config uses an `overrides` directory key. Documentation explains how they are structured (one JSON per breakpoint) [180]. Consistent usage.

Engine – The internal class that runs validation. Mentioned in architecture and API docs. For users writing custom plugins or using DCV programmatically, Engine is the interface to get/set token values and attach plugins. The term is used in developer docs, not much for end-users. It's consistent (there's only one "Engine").

Plugin – The mechanism by which each constraint type is implemented. Documentation refers to built-in plugins (MonotonicPlugin, etc.) in Architecture doc [181] [92], and mentions you can add custom ones. Synonymous with constraint type implementation. No issues, just an internal term.

In summary, most domain terms are well-defined in context, but **some naming inconsistencies exist**: - *"failOn: off" vs "failOn: never"* – clearly a rename that wasn't propagated everywhere [13] [14]. - *"policy" vs "themes/constraints"* – conceptual vs actual config term; could confuse users if not explained that a "policy file" is basically a constraints JSON. - *"theme"* – overloaded term (should perhaps be replaced with "constraints directory" in user-facing language to avoid misinterpretation). - Minor JSON field naming differences (kind vs ruleId, etc.) – should unify across docs.

Chronology Map

Understanding DCV's documentation chronologically helps identify lingering old ideas:

- **Early Stage (concept & pre-implementation):** The "Decision Themes" prior-art document represents very early thinking. It outlines a grand vision of a two-tier theming system (with DCV as a validator at the end). This likely predates the actual DCV tool or coincided with its inception. At this stage, key ideas were: **post-compute validation** (validate what you actually ship, not intermediate states) and **deterministic outputs with receipts** for governance [167]. It also introduced the term "themes" in context of design systems (Visual vs Decision themes) and dual-namespacing. This thinking influenced DCV's design: for example, DCV was built to not assume any specific design tool, to work purely on resolved values, and to generate receipts (all of which it does). However, some notions from this phase are not implemented in DCV itself (DCV doesn't handle computing the EffectiveConfig – that's left to external tools; DCV doesn't know about `--ui` vs `--ds` namespaces, it just validates whatever tokens you give). So the **older idea of "Decision Themes" lives on indirectly** – DCV's documentation still carries the torch of being engine-agnostic and focusing on final values, as set out in that doc, but the specifics of dual namespaces are not relevant to DCV's user docs.

- **Mid Development (initial tool creation):** As the DCV code was written (likely in mid-2025), the immediate docs were probably the README and maybe a basic config reference. The root README likely started as the primary doc, pitching the need for "constraints, not conventions" and showing basic usage. During this time, only a subset of constraint types might have existed (perhaps Monotonic and WCAG first, then Threshold/Lightness, and Cross-Axis last given its complexity). The **Configuration.md** file in root looks to have been written around here: it's detailed, but uses an older name for `failOn("never")`, suggesting the CLI semantics weren't finalized yet. It doesn't mention cross-axis (it enumerates keys like wcag and order, presumably before cross-axis rules or built-in thresholds were fully formed). Also, the CLI help in README (e.g. an example of `--policy` flag usage) might reflect an earlier interface idea (like grouping constraints as a single policy file) that isn't prominent in later docs.

- **Late Development (pre-release hardening):** As DCV approached a 1.0 release, significant efforts went into **expanding documentation and aligning it with the final feature set**. The creation of the entire `/docs` folder, all dated around the same time, indicates a push to provide a complete user guide. The defensive publication *DCV Post-Compute Validation & Receipts* was written on Nov 7, 2025 [3] – likely just before release – codifying the design formally (perhaps for intellectual property reasons, but also serving as a technical whitepaper). This document solidifies the architecture that matches the implementation (it reads almost like an **architecture handbook**, and indeed the Architecture.md mirrors many sections of it in a more user-friendly tone). It's worth noting that by this time, new features like Cross-Axis constraints had been added (the doc includes them), and performance considerations (like parallel validation) were implemented (mentioned in Architecture doc). So late-stage docs incorporate these. The cross-axis guide's enthusiastic tone (with an emoji ⚡ and real examples) suggests it was a recent addition considered a highlight of the release.

- **After Release (ongoing tweaks):** Post 1.0, smaller adjustments to docs can be seen (for example, the JSON output format might have been refined – the presence of both `kind`/`severity` vs `ruleId`/`level` in different docs implies that during final QA, they changed some field names or CLI flags). The **AI guide** was likely added right after or alongside release to help answer user questions through AI channels, indicating forward-thinking support. The duplications (root vs docs) likely exist because the team didn't want to remove the old docs until the new ones were proven, or simply to cater to both in-repo readers and wiki readers. Now that the dust has settled, some of those older docs (root CONFIGURATION.md especially) feel like remnants of an **earlier documentation phase**.

Older ideas still present vs new directions: One example is the terminology of **"themes"**. Historically, the project's conceptual framework (Decision Themes) used that term. The code and config still use "themes" as a directory name for constraints, which is a bit misaligned with how the tool is positioned now (it's not actually validating multiple visual themes; it's enforcing constraints). This is an artifact of earlier thinking where maybe constraints and themes were part of one ecosystem. Now DCV is focused purely on constraints, and calling them "constraints" or "policies" might be clearer – yet the term "themes" persists in file paths and some docs. It's an old idea (the design tool's theme) bleeding into the implementation lexicon.

Another area: the **policy profile concept** – early on, it seems there was an idea to allow swapping constraint sets easily (hence "--policy aa.json"). While DCV can do that by pointing to different folders or files, the new docs emphasize having a directory of constraints and possibly enabling/disabling via config. The CLI example in README for `--policy` might represent an older approach that isn't foregrounded anymore. The newer docs don't explain `--policy` usage at all, suggesting that feature either was deprioritized or replaced by config-driven selection. Yet the example remains, which could mislead users if it's not actually the recommended path.

In summary, the documentation shows a project that started with a very **big picture vision (theming systems, decision logic)** and distilled it into a practical tool solely for validation. Most of the older conceptual stuff has been translated into concrete features, but occasionally the **vocabulary and emphasis in docs reveal the evolution** (e.g., "themes" directory name from the theming concept, mention of policy packs, etc.). The current direction as evidenced by recent docs is a pragmatic validator tool for design tokens, and the documentation mostly aligns with that, with just minor leftover references to the broader vision.

Red Flags in Documentation

- **Duplicate or Conflicting Sources:** The existence of two configuration guides (root CONFIGURATION.md vs docs/Configuration.md) is a prime red flag. They contain nearly the same information but with slight discrepancies. For example, the accepted values for the `failOn` setting differ ("off" vs "never") [13] [14], and the root one doesn't document newer options like `summary`. This can confuse users as to which is correct. Similarly, overlapping content between README and docs/Home or docs/Getting-Started means if updates are made in one place and not the other, inconsistencies will arise.

- **Out-of-sync CLI instructions:** There are inconsistencies regarding CLI flags and output schemas:

- The main README and AI Guide use `--format json` to get JSON output [66], but CLI.md documents `--summary json` [67]. If both flags exist or one replaced the other, it's not clearly communicated. This inconsistency will trip up users trying to follow examples. It suggests documentation wasn't fully updated after a change in the tool's interface.
- JSON output format naming differences: The CLI reference's sample JSON uses keys like `"severity"` and `"kind"` [63], whereas the JSON-OUTPUT spec and actual code prefer `"level"` and `"ruleId"` [117]. This is a red flag because it may indicate the CLI doc example is outdated, and if a user writes scripts expecting the wrong field names, it could break. All docs need to consistently reflect the actual output structure.

- Another subtle inconsistency: the CLI doc calls the JSON flag `--summary <format>` but the JSON-OUTPUT and usage examples treat `--format` as the flag. It's possible both flags exist for different commands (maybe `validate` expects `--summary` while `graph` uses `--format`?), but that's not clarified. This will be confusing unless resolved or explained.

- **Terminology confusion – "themes" and "policy":** As noted, the use of "themes" to mean constraint files is potentially misleading. A user reading "Path(s) to constraint/theme files" [173] might wonder "what's a theme file? I thought we're talking about constraints." If they come from design tokens background, "theme" means something else (like a theming context). Similarly, the quick start instructs making a `themes/` folder for constraints, which might seem odd. The docs do not explicitly clarify why it's called themes. This inconsistency in terminology is a red flag as it can lead to misunderstanding. The documentation should either adopt "constraints directory" language or explain the historical reason, to avoid ambiguity.

Likewise, the README's mention of "policy profile (e.g., AA)" [172] and presence of `themes/policies/aa.json` example implies one can have a "policy" file. But nowhere in docs/Constraints or docs/Configuration is "policy" formally defined or the structure of a combined policy file explained. If a user tries to find what an AA policy JSON looks like, they might search in vain (unless they dig into `themes/policies/aa.json` in the repo which presumably exists with some predefined constraints). Promising something ("policy profiles") without clear documentation is a red flag. It may be a leftover idea that isn't fully fleshed out in docs.

- **Threshold constraints documentation gap:** In the Constraints guide, all constraint types except Threshold are demonstrated with JSON config examples. Threshold only shows a TypeScript snippet, which might lead users to think they *cannot* configure custom thresholds. It references WCAG and Apple HIG 44px, but doesn't say "how do I enforce a different min or add my own threshold rule?" If threshold rules are hardcoded, that should be stated; if they are configurable, an example JSON should be given. The absence of this is a documentation red flag (unclear feature usage).

- **Unimplemented promises:** So far, the docs don't blatantly describe features that don't exist, which is good. One possible minor one: the Decision Themes paper and even some phrasing in README might suggest integration with design tools or receipts used in CI. DCV does produce receipts, but if a user expects more (like an included way to compute tokens), they won't find it. However, the docs never claim DCV computes tokens (it explicitly says it's not a token processor but a validator [182]). So not exactly a broken promise, just something to keep clear.

- **Wiki sync requirement:** Because the docs are duplicated on the wiki, there's a maintenance red flag: the WIKI-SETUP notes show the API page hadn't been copied yet at one point [143]. If the wiki is public and incomplete or outdated relative to `/docs`, users might get the wrong info. Relying on manual sync (copy-paste) between docs and wiki is error-prone. Any divergence would be a documentation consistency problem. This is a process red flag – though the team is aware (hence that doc) – it's worth highlighting.

- **Scattered advanced info:** Some advanced or niche info is only findable deep in docs or code comments. For instance, the CLI test (in code) revealed the existence of a `--mapper` for build and how patch works, but fortunately CLI.md does mention it. If there were any features only documented in code comments (none obvious from our scan), that would be a red flag. It appears most features have at least a mention in the docs, which is good. The **adapters information** might be an exception: README referenced an `adapters/README.md` that doesn't actually seem present in the repo (broken link) [183]. That means documentation about supported input formats (Style Dictionary vs Tokens Studio nuances) might be missing or incomplete – a red flag for users who need that. The examples cover some of it, but a centralized explanation is absent (possibly that missing file was planned).

- **Minor content errors:** e.g., in WIKI-SETUP it says “Already copied the docs to the Wiki!” [184] which might just be phrasing, but if someone reads it in the repo they might be confused (“did I? who copied?” – it’s written as if the action is done). Not critical, but slightly unclear wording.

Overall, the red flags revolve around **inconsistency and duplication**. The actual substance of the docs is high-quality; the main risk is some users reading one piece (like the root README or wiki) that hasn’t been updated and getting outdated instructions. Cleaning those up will resolve most issues.

Missing Documentation

After reviewing, a few gaps where the code or functionality isn’t fully explained in docs come to light:

- **Adapters and Input Formats:** The README promises that DCV supports multiple token formats and even points to an `adapters/README.md` [183] for details, but that file either doesn’t exist or is not easily found. There is no dedicated section in the docs explaining how tokens from *Tokens Studio JSON* or *Style Dictionary* or *DTCG (W3C Design Token format)* are handled. Experienced users might infer that DCV auto-detects or requires certain fields (`$value`, etc.), but novices could use guidance. For example, how does DCV treat the `"type": "color"` metadata in Style Dictionary input? The Architecture doc shows a snippet of normalization [82] [185], but there’s no user guide on “Using DCV with Popular Token Formats”. This is a documentation gap – presumably the missing adapters doc would have covered it. Providing a table of supported input JSON patterns or a brief guide would help.

- **Custom Constraint Plugins:** The architecture doc hints at extension via custom plugins [103], but nowhere in user docs or API reference does it explicitly walk through how to write and plug in a custom constraint. If the intent is to allow it (the Engine class is exposed, and `Engine.use(plugin)` is part of the API [95] [186]), a short guide or example would be valuable. As of now, a developer would have to reverse-engineer from built-ins. So a “How to add your own constraint type” section is missing.

- **Using DCV in CI/automation beyond basic**: While the AI Guide and JSON-OUTPUT doc give some pointers (like a GitHub Actions snippet [130]), the docs don’t explicitly mention how one might integrate DCV in, say, a pre-commit hook or a CI pipeline in detail. They do mention `--fail-on off` for CI usage in a couple places, which is good. This missing piece is minor, as advanced users can figure it out, but a dedicated “CI Integration” page could be nice (the AI guide touches on it in the context of AI Q&A, but not every user will see that).

- **Policy file schema:** If DCV indeed supports a single JSON that contains multiple constraint sections (as implied by the term “policy” and the ability to define constraints in `dcv.config.json` under a `"constraints"` key [187]), then documentation should show the structure of such a file. E.g., can one create `myPolicy.json` with `{ "constraints": { "wcag": [...], "threshold": [...], "rules": [...] } }` and feed it to DCV? The docs don’t provide a complete example of that, though pieces are scattered (Getting Started shows separate files, Configuration shows one example of inline constraints with just WCAG [188]). A clear example of a combined policy JSON (maybe in the themes/policies folder in repo) exists but isn’t in the docs. Including that would fill the gap.

- **Threshold constraints usage:** As noted, how to specify custom threshold rules is not shown. Are the threshold rules hard-coded (like always checking 44px)? Or can the user supply a JSON file similar to how WCAG is supplied? If yes, an example like `themes/thresholds.json` with a custom rule should be documented. Otherwise, at least state “DCV automatically checks certain thresholds (44px touch) when strict mode is on” (if that’s the case). Right now it’s ambiguous.

- **Clarification of default constraints:** Does DCV do anything out-of-the-box without any constraint files? This isn’t explicitly stated. The docs say “No constraints found” as an error if you don’t provide any [189]. So presumably it does nothing unless you give constraints (except maybe the built-in threshold if strict? but not sure). This is fine, but maybe worth stating clearly: “DCV will only enforce what you specify, except possibly in strict mode which enables additional built-ins.” If strict mode indeed enables threshold and maybe any other implicit rules, that needs documentation (strict is mentioned but not fully explained on what it does beyond “fail on any issue” – what “issues” specifically?). “Strict” might refer to treating warnings as errors rather than adding checks. It’s a bit unclear.

- **Example-driven guides for advanced scenarios:** The documentation is comprehensive on reference, but could be missing a narrative “tutorial” for a multi-breakpoint project or using DCV with an existing design system. For instance, an end-to-end example: “Integrating DCV into an existing Style Dictionary pipeline” or “Validating multiple themes (light/dark) with DCV.” The pieces to do these are present, but not assembled in a guide. While not strictly necessary, these kinds of docs are often helpful to users in real scenarios.

In summary, most of DCV’s functionality *is* documented, but the **integration points (adapters)** and **extensibility (custom plugins)** are where documentation is scant. Also, anything that was partially implemented (like the notion of built-in constraints in strict mode, if any) could use explicit mention.

Recommended Canonical Documentation Set

To reduce confusion and ensure maintainability, I recommend the project **converge on a single, coherent set of docs** and deprecate the rest. The following should be considered the “source of truth” going forward:

- **The `/docs` folder contents should be the canonical documentation** for users. Specifically:
- *Getting-Started.md, Constraints.md, CLI.md, Configuration.md, Architecture.md, API.md, JSON-OUTPUT.md* should be maintained and updated with each feature/fix. These cover all essential aspects (tutorial, usage reference, internals, integration).

- The *Home.md* in docs (which doubles as wiki Home) can serve as a portal, but it should not contain content that isn’t in the above files. Ideally, Home.md just links to sections in the main docs or provides a high-level intro. It’s fine to keep, just ensure it’s synced and doesn’t diverge in messaging (currently it repeats some text from README – that duplication is okay if it’s identical, but any changes need to be mirrored).

- **Root README.md:** This should be **trimmed or refocused** to avoid duplicating the docs. I suggest keeping the README as a short introduction and quick-start, then pointing to the detailed docs. For example, the README can have the tagline, installation instructions, a very brief example, and then “**See the [Documentation]** for full guides and reference.” This way, there’s one place to update detailed instructions (in /docs), and the README stays consistent by delegation. Currently, the README has extensive content that overlaps with docs – by paring that down, you eliminate one source of divergence. (Important: if README is also the npm page, it should still contain enough info to hook users, but it can reference the docs for depth.)

- **Deprecate/remove root CONFIGURATION.md:** Its content is now fully encompassed by docs/Configuration.md (with newer info to boot). To avoid anyone reading outdated advice, consider removing it from the repo or at least adding a big notice at top: "This guide is deprecated – see docs/Configuration.md". The best approach is to fold in any missing bits (if any) from it into the docs version and then delete or clearly mark it. The docs/Configuration.md should be the one true config reference.

- **Use one terminology consistently:** In all canonical docs, choose either "off" or "never" for `failOn` (presumably "off" since it appears in newer docs), and update any stragglers. Do the same for JSON field names (`ruleId` vs `kind`, etc.) so that API, CLI, and JSON docs all speak the same language. This may involve updating CLI.md and API.md examples to match JSON-OUTPUT.md. Once done, the corrected docs in /docs will be canonical for how to use flags and interpret output.

- **Clarify the "Policy/Theme" wording:** In the canonical set, decide on how to present this. Perhaps call them "constraint files" or "constraint sets" instead of loosely saying "themes" everywhere. You could introduce the term "Policy" in the user docs as a concept: e.g., "A policy is just a collection of constraint definitions. By default, DCV loads all *.json files in the `themes/` folder as your active policy." Then consistently call them constraint files or policy files. The key is to make sure users don't confuse "theme" with something else. The docs could even recommend using a different folder name if they prefer (since the path is configurable). In any case, the canonical docs should not leave that ambiguity hanging. Given this is more wording than content, it's about editing those docs.

- **Prior-art docs:** These should be treated as separate reference material. They are valuable, but not part of the day-to-day user guide. I'd suggest linking to them under an "Architecture Notes / Background" section in Architecture.md or the README, for those interested in the theoretical background. They don't need to be updated with each release (since they are snapshots), but the canonical docs can note: "For a deeper discussion of DCV's design rationale, see Prior Art: Post-Compute Validation." That way, they're acknowledged but clearly secondary. The canonical truth of "how the tool works now" remains in the main docs.

- **Examples and adapter info:** Incorporate the missing adapters documentation into the canonical set. This could be a new page (e.g., docs/Adapters.md) or just a section in either Configuration or Getting Started. It should list the supported input formats and any special notes (like "If you use Style Dictionary, DCV will interpret its file structure like so…", etc.). Since README already attempted to link to adapters/README, fulfilling that promise in the docs set is important. Once that exists, that page becomes canonical for format support.

- **Wiki vs Repo docs:** To avoid the burden of dual maintenance, it might be wise to choose one as primary. If the repo (/docs) is primary (which it should be, since it's versioned with code), ensure that all canonical changes happen there and then occasionally sync to wiki. The wiki is essentially a published view of /docs. We don't want a scenario where someone updates the wiki and not the repo docs or vice versa. So making it clear to contributors: "Edit the markdown in /docs, not directly on the wiki," will help. In the canonical set, consider removing WIKI-SETUP.md from user-facing docs (it's maintainers-only). It could be moved to a /docs/internal or just an internal note outside of main docs. This keeps canonical user docs focused.

- **Single source for CLI and API reference:** The CLI.md and API.md should remain separate (different audiences), but ensure they are kept consistent. If needed, cross-reference them or generate parts from code. For example, if flag descriptions can be pulled from the CLI help output, that could reduce drift. For now, the action is to manually align them as discussed.

- **Focus on /docs as the one truth:** After these adjustments, the idea is that **any user question should be answerable by pointing to a /docs page**. If something is only in README or an example, consider migrating that info. For instance, the example READMEs contain specifics (like how to run those scenarios) – if any important general knowledge is in there, copy it into a guide. Otherwise, it's fine for them to remain as isolated how-tos.

Recommended set moving forward: - README (concise intro + link to docs) - `/docs/`: - Home (or Overview) - Getting Started - Constraints Guide - CLI Reference - Configuration Reference - JSON Output Reference - API Reference - Architecture Overview - (possibly Adapters/Formats guide) - Developer docs: CONTRIBUTING, RELEASE (these can remain in root or in docs, but are separate from user docs)

Everything outside this list (duplicate guides, outdated terminology, etc.) should be pruned or merged in. This will give DCV a clear, **canonical documentation hub** that is easier to maintain and for users to navigate, ensuring that as the code evolves (e.g., DCV 1.1, 2.0), there is one place to update each piece of info and no mixed messages.

---

1 2 142 143 144 145 146 184 WIKI-SETUP.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/WIKI-SETUP.md

3 147 148 149 150 151 152 153 154 155 156 157 158 182 DCV-Post-Compute-Validation-and-Receipts.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/prior-art/DCV-Post-Compute-Validation-and-Receipts.md

4 159 160 161 162 163 164 165 166 167 168 Decision-Themes-Deterministic-Compute-and-Dual-Namespaces.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/prior-art/Decision-Themes-Deterministic-Compute-and-Dual-Namespaces.md

5 19 20 README.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/README.md

6 21 22 23 81 Home.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/Home.md

7 8 9 10 11 12 13 180 CONFIGURATION.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/CONFIGURATION.md

14 68 69 70 71 72 73 74 75 76 77 78 79 80 173 178 187 188 Configuration.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/Configuration.md

15 16 CONTRIBUTING.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/CONTRIBUTING.md

17 18 RELEASE.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/RELEASE.md

24 172 175 183 README.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/README.md

25 26 27 28 29 30 31 32 33 34 35 36 189 Getting-Started.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/Getting-Started.md

37 38 39 40 41 42 43 44 45 46 47 48 49 169 179 Constraints.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/Constraints.md

50 51 52 53 54 55 56 57 58 59 60 61 62 63 65 67 176 CLI.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/CLI.md

64 117 118 119 120 121 122 123 124 125 126 127 128 129 130 170 177 JSON-OUTPUT.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/JSON-OUTPUT.md

66 131 132 133 134 135 136 137 138 139 140 141 AI-GUIDE.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/AI-GUIDE.md

82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 181 185 186 Architecture.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/Architecture.md

104 105 106 107 108 109 110 111 112 113 114 115 116 171 API.md
https://github.com/CseperkePapp/design-constraint-validator/blob/2ce2638d7c46705eea02483361498e61ac79a71e/docs/API.md

174 cli.test.ts
https://github.com/CseperkePapp/SingleLarissa/blob/3bfc775ff0de90710f56882885480889228826d0/test/cli.test.ts