DocsGen Documentation

Combined PDF of the complete Markdown docset (Base, Architecture, UI, Appendix, Diagrams).

Format: rendered Markdown (headings, lists, tables, code blocks).

Source: MODULE-docsgen.zip (45 files).

Generated: 2025-12-14 22:46 UTC

Note: Code blocks and ASCII diagrams are preserved as monospaced blocks.

Table of Contents

File: docsgen.md

DocsGen

Purpose

DocsGen is a **phase-gated documentation generation plugin** that produces complete, implementation-ready plugin documentation from structured inputs.

It is designed for environments where documentation must be **auditable**, **reproducible**, and **safe to approve**, even when large language models are used for drafting. DocsGen enforces this by separating concerns into clear planes (UI, orchestration, intelligence, deterministic validation) and by treating every generated file as an immutable artifact that must pass validation gates before the run can continue.

DocsGen is intended to document other plugins (and itself) using a fixed 43-file docset structure and a six-phase execution model (Preflight + Phases 1–5).

Glossary

Run One end-to-end execution of the documentation pipeline for a single plugin, producing immutable artifacts.

Phase A bounded generation step with explicit inputs, outputs, and a deterministic validation gate.

Gate A deterministic checkpoint that must pass (or be explicitly approved) before moving to the next phase.

Artifact Any stored output created during a run (core documentation files, validation reports, planned file lists, bundles).

Core Documentation File One of the 43 numbered Markdown files that constitute the deliverable documentation set.

Generation Artifact A supporting file produced during generation (planned file list, terminology index, validation reports). These are not part of the deliverable bundle.

Strictness A validation policy mode: strict requires full enforcement; lenient relaxes specific non-core rules while keeping core guarantees.

Review Mode A run mode: interactive stops after each phase for approval; batch runs all phases without stopping.

Orchestrator The control-plane service that owns the run state machine, artifact registry, and validation decisions.

Agent A Foundry-hosted LLM worker that generates Markdown content under orchestrator control.

Tech Split

Layer / Plane	Owns	Must Not Own
Experience Plane (Blazor UI)	Uploads, run configuration, progress display, approvals, downloads	LLM calls, file generation, validation logic
Control Plane (Orchestrator API)	Phase state machine, gates, artifact registry, validator execution, streaming events	UI rendering, LLM reasoning, non-deterministic judgment
Intelligence Plane (Foundry Agents)	Drafting Markdown content, optional narrative issue summaries	Phase control, artifact overwrite decisions, truth authority
Deterministic Plane (Validators + Storage)	Link checks, placeholder scans, schema enforcement, scoring, bundling, artifact persistence	LLM judgment, implicit regeneration

---

Guiding Principle

LLMs generate content. Deterministic systems decide truth.

Scope Definition

In Scope

- Structured, phase-based documentation generation
- Deterministic validation gates between phases
- Artifact immutability and run auditability
- Interactive approvals and partial regeneration
- Tool/API surface for deterministic operations (read/write artifacts, validate, bundle)

Out of Scope

- Publishing documentation to external portals
- Collaborative multi-user editing and merge conflict resolution
- Runtime business logic for unrelated plugins
- Replacing deterministic validators with LLM judgment

Non-Goals

DocsGen is intentionally designed to avoid “silent correctness.”

- DocsGen **does not** auto-fix validation failures by regenerating prior approved content.
- DocsGen **does not** treat an LLM response as valid documentation unless deterministic gates pass.
- DocsGen **does not** execute or validate host application runtime behavior (it validates documentation only).

- DocsGen **does not** require the UI to understand file semantics; it only renders orchestrator state.

Success Criteria

DocsGen is considered successful when:

- A strict run produces **43 core documentation files** with **0 broken internal links**.
- No placeholder strings remain in any core file.
- Terminology is consistent across Base, Architecture, UI, Appendix, and Diagrams.
- A failed phase generates an issues report and stops without overwriting approved artifacts.
- Partial regeneration can target only the affected files and re-run gates deterministically.

DocsGen – Documentation Index

This document is the canonical entry point for the DocsGen documentation set.

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

File: docsgen-00-base-index.md

Base Layer

This section defines the **conceptual foundation** of DocsGen. It explains terminology, core abstractions, invariants, and canonical schemas without assuming a specific hosting runtime or cloud provider implementation.

- **Base Vision**

Why DocsGen exists, what it optimizes for, and what it deliberately avoids.

- **Base Concepts**

Canonical terminology: runs, phases, gates, artifacts, strictness, review mode, and source-of-truth rules.

- **Base Models**

The domain model: Run, PhaseExecution, Artifact, ValidationResult, Approval, and policy objects.

- **Base Phase Types**

The six phases (Preflight + 1–5) and what each phase must produce.

- **Base Planning**

Planned file lists, docset profiles, numbering invariants, and link authority rules.

- **Base Execution Model**

The state machine and how deterministic gates control progression.

- **Base Gate Protocol**

The canonical gate algorithm: generate → validate → report → stop/approve.

- **Base Integrations**

Conceptual integration points: Foundry agents, tool APIs, artifact storage, and host boundaries.

- **Base Examples**

Example runs (interactive vs batch), partial regeneration, and typical outputs.

- **Base Synchronization**

Immutability, versioning rules, audit trails, and regeneration policies.

- **Base Schema**

Canonical JSON schemas for runs, artifacts, issues, and gate reports.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-01-base-vision.md

Base Vision

What DocsGen Is

DocsGen is a documentation system designed for environments where documentation must be:

- **Reviewable** (humans can approve at phase gates)
- **Deterministically validated** (pass/fail rules are code, not LLM judgment)
- **Reproducible** (runs can be replayed, compared, and audited)
- **Safe to iterate** (partial regeneration without content drift)

DocsGen assumes that large language models can draft content quickly, but it refuses to treat drafts as final until deterministic validators prove that constraints are satisfied.

The Problem It Solves

AI-generated documentation often fails in predictable ways:

- Links drift and break as the docset grows
- Terms change between sections, causing ambiguous implementation requirements
- Early outputs get overwritten silently while later phases are being generated
- Validation is informal (“looks good”) instead of deterministic
- The UI becomes the de facto state machine, creating hidden behavior

DocsGen solves these by making the orchestrator the single source of truth, enforcing immutability, and requiring explicit user approvals when configured.

Design Philosophy

Deterministic Gates as the Authority

The only thing that can advance a phase is a passing validation gate (or a recorded override when the policy allows it). The orchestrator records why each gate passed.

No Silent Regeneration

Once a phase has been approved (explicitly or implicitly, depending on mode), its core documentation artifacts must not be overwritten without an explicit regeneration action.

Short, Chunked Agent Work

Agents must be invoked in bounded steps (plan, then generate file-by-file), rather than producing multi-file monoliths that are more likely to time out and drift.

Inputs Win Over Prose

Structured inputs (plugin requirements JSON) outrank all other sources, and must be treated as the canonical truth when there is conflict.

Boundaries (What DocsGen Must Not Become)

DocsGen is not:

- A free-form “documentation writer” without structure
- A monolithic agent loop that decides correctness
- A system where a UI click triggers regeneration of previously approved files
- A platform that depends on human memory for which rules were enforced

---

Success Definition

DocsGen succeeds when it produces documentation that is:

- Strict-mode valid (no broken internal links, no placeholders, terminology consistent)
- Implementation-grade (engineers can build from it without guessing)
- Traceable (every file can be traced to a run + phase + inputs + validation results)

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-02-base-concepts.md

CONCEPTS

Mental Model

DocsGen produces plugin documentation by running a **phase-gated pipeline**.

- A **Run** is the unit of work.
- Each run advances through **Phases** (Preflight + 1–5).
- Each phase outputs immutable **Artifacts**.
- Each phase ends with a deterministic **Gate**.
- A gate produces a report and either passes, fails, or awaits approval.

DocsGen is intentionally designed so that *progress is explicit* and *truth is deterministic*.

Core Terms (Canonical)

Run

A run is an execution instance that produces artifacts. A run has:

- configuration (review mode, strictness)
- a current phase
- a lifecycle status (running, awaiting approval, failed, complete)

Runs are not overwritten; they are archived and diffable.

Phase

A phase is a bounded generation step that:

- has known inputs
- produces explicit outputs
- can be re-run without regenerating unrelated artifacts

Phases exist to prevent drift and to make validation manageable.

Gate

A gate is a deterministic evaluation executed after a phase finishes generation.

A gate produces:

- a structured issue list (file, rule, message)
- a human-readable report
- a pass/fail decision

Artifact

An artifact is a stored file with metadata:

- logical path (run + phase + filename)
- content hash
- type classification (core doc, report, input, bundle)
- provenance (agent, tool, user)

Artifacts are immutable under the default policy.

Core Documentation File

A core documentation file is one of the 43 numbered Markdown files defined by the docset structure. These are the deliverable files bundled at the end of the run.

Generation Artifact

A generation artifact is a support file produced during the run (planned file list, terminology index, validation reports). These are useful for review and auditing, but are not part of the deliverable bundle.

Source-of-Truth Precedence

When inputs conflict, follow this precedence:

1. Plugin requirements JSON (canonical truth)
2. Host architecture constraints (runtime truth)
3. Reference docset conventions (structure truth)
4. Natural language instructions (context only)

This is a rule enforced by the orchestrator and validators, not by “best effort” prompting.

Global Content Rules

No Placeholders

Final outputs must not include placeholder tokens, ambiguous “fill this later” content, or unresolved template language.

Link Integrity

All internal links must point to existing core documentation files and use relative paths.

Terminology Consistency

Terms defined in Base Concepts remain canonical and must not be redefined differently elsewhere.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-03-base-models.md

Base Models

Overview

DocsGen is easiest to reason about when its core model is explicit. This file defines the canonical domain objects and how they relate.

The model is designed to support:

- deterministic validation
- artifact immutability
- partial regeneration
- interactive approvals
- auditability across runs

Run

A **Run** is the top-level container for everything produced during generation.

Responsibilities

- stores run configuration (review mode, strictness)
- tracks current phase and lifecycle status
- anchors the artifact namespace

Canonical Fields

json

```
{
  "runId": "run_2025_12_14_0001",
  "pluginSlug": "docsgen",
  "reviewMode": "interactive",
  "validationStrictness": "strict",
  "currentPhase": 2,
  "status": "running",
  "startedUtc": "2025-12-14T00:00:00Z",
  "completedUtc": null,
  "artifactsManifestId": "manifest_run_2025_12_14_0001_v3"
}
```

PhaseExecution

A **PhaseExecution** is a recorded attempt of a single phase. A phase may have multiple executions if it was explicitly regenerated.

Responsibilities

- records the agent used
- records inputs and outputs (artifact references)
- records validation results and approval decisions

Canonical Fields

json

```
{
  "runId": "run_2025_12_14_0001",
  "phaseNumber": 2,
  "executionId": "phase2_exec_001",
  "agentId": "base_layer_generator",
  "inputs": [
    "input/plugin-requirements.json",
    "phase1/docsgen.md",
    "phase1/docsgen-00-base-index.md"
  ],
  "outputs": [
    "phase2/docsgen-02-base-concepts.md",
    "phase2/docsgen-11-base-schema.md"
  ],
  "validationResult": {
    "status": "pass",
    "issues": [],
    "reportArtifact": "phase2/02-phase2-validation.md"
  },
  "approval": {
    "required": true,
    "status": "pending",
    "approvedBy": null,
    "approvedUtc": null
  }
}
```

Artifact

An **Artifact** is a stored file plus metadata.

Responsibilities

- enable immutability and audit
- support re-validation without regeneration
- allow diffing across runs

Canonical Fields

json

```
{
  "logicalPath": "run_2025_12_14_0001/phase2/docsgen-02-base-concepts.md",
  "contentHash": "sha256:9b3a...e12f",
  "sizeBytes": 3242,
  "artifactType": "CoreDoc",
  "createdBy": "agent",
  "createdUtc": "2025-12-14T00:12:43Z"
```

```
}
```

Artifact Types (Canonical)

- **Input** – uploaded inputs
- **CoreDoc** – one of the 43 deliverable files
- **Report** – validation report or issues report
- **Bundle** – final zip
- **Plan** – planned file list
- **Index** – terminology index and similar registries

ValidationResult and Issue

A **ValidationResult** is the deterministic output of a gate.

json

```
{
  "status": "fail",
  "summary": "2 broken links, 1 forbidden placeholder token",
  "issues": [
    {
      "ruleId": "links.internal.resolves",
      "severity": "error",
      "file": "docsgen-26-ui-index.md",
      "message": "Link target does not exist: docsgen-38-ui-examples.md",
      "location": { "line": 17, "column": 5 }
    },
    {
      "ruleId": "placeholders.forbidden",
      "severity": "error",
      "file": "docsgen-03-base-models.md",
      "message": "Forbidden placeholder token detected.",
      "location": { "line": 4, "column": 1 }
    }
  ]
}
```

Invariants

- A gate “pass” means zero error-severity issues.
- A gate “fail” means at least one error-severity issue.
- Warnings may be allowed depending on strictness policy.

Policy Objects

Policies are deterministic configuration objects owned by the orchestrator.

ValidationProfile

Defines which validators apply and how strict they are.

RegenerationPolicy

Defines whether overwrites are allowed and under what explicit run modes.

DocsetProfile

A **DocsetProfile** determines which files are required and which can be optional.

Examples:

- Full strict profile (all 43 files required)
- Lenient profile (diagrams optional; coverage threshold reduced)

Docset profiles prevent “special cases” from creeping into prompts by making requirements explicit.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-04-base-phase-types.md

Base Phase Types

Overview

DocsGen uses a six-step progression:

- **Phase 0**: Preflight (blocking)
- **Phase 1**: Foundation (root + indices + planned files)
- **Phase 2**: Base Layer (conceptual truth + terminology index)
- **Phase 3**: Architecture Layer (implementation truth)
- **Phase 4**: UI & Appendix (user-facing + project mapping)
- **Phase 5**: Diagrams & Cross-Validation (global consistency)

Each phase is defined by:

- required inputs
- required outputs
- a deterministic validation gate

Phase 0 – Preflight (Blocking)

Goal

Confirm that required inputs exist and are internally consistent before any documentation files are generated.

Inputs

- plugin requirements JSON
- host project structure reference (zip or tree)
- documentation formula
- example docset zip (structure conventions)

Outputs

- Preflight summary (if passing), or preflight questions (if failing)

Gate Behavior

Preflight failure halts the run in all modes and all strictness levels.

Phase 1 – Foundation

Goal

Generate the structural skeleton that all later phases depend on.

Required Outputs

- Root doc file (entry point)
- Base index
- Architecture index
- UI index
- Appendix index
- Planned file list (generation artifact)

Gate Focus

- all required files exist
- no placeholders
- navigation links resolve

Phase 2 – Base Layer

Goal

Define the conceptual and schema-level truth of the plugin without implementation details.

Required Outputs

- 11 base layer files (01–11)
- terminology index (generation artifact)

Gate Focus

- terminology consistency
- schema completeness
- internal link integrity

Phase 3 – Architecture Layer

Goal

Document how the plugin is implemented and integrated.

Required Outputs

- 13 architecture files (13–25)

Gate Focus

- references to base layer schema are consistent
- no new entities introduced
- integration and dependency rules are explicit

Phase 4 – UI & Appendix

Goal

Document user-facing workflows and map documentation to project structure.

Required Outputs

- 12 UI files (27–38)
- appendix files (40–42)

Gate Focus

- UI validation messages match canonical rules
- appendix correctly maps components/services/models to docs

Phase 5 – Diagrams & Cross-Validation

Goal

Produce diagrams and run global cross-validation across the entire docset.

Required Outputs

- diagrams file (43) in strict mode
- final validation report
- bundle zip containing all core files

Gate Focus

- full docset link integrity
- cross-layer schema/terminology consistency
- completeness threshold (coverage scoring)

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-05-base-planning.md

Base Planning

Why Planning Exists

DocsGen treats the planned file list as the **link authority** for the entire run.

Without planning, generation tends to drift:

- index files link to non-existent targets
- optional files are referenced even when excluded
- file naming diverges between phases

Planning solves this by forcing an explicit list before content generation begins.

Docset Structure (43 Core Files)

DocsGen uses a fixed 43-file deliverable structure:

- Root entry file: `docsgen.md`
- Base index and base files: `docsgen-00` through `docsgen-11`
- Architecture index and architecture files: `docsgen-12` through `docsgen-25`
- UI index and UI files: `docsgen-26` through `docsgen-38`
- Appendix index and appendix files: `docsgen-39` through `docsgen-42`
- Diagrams: `docsgen-43`

The numbering is stable so that:

- validators can be phase-aware by file ranges
- humans can navigate consistently across plugins
- partial regeneration can target predictable regions

Planned File List (Link Authority)

A planned file list is a generation artifact produced at Phase 1 and treated as authoritative for link validation.

Example (DocsGen full docset):

json

```
{
  "version": "1.0",
  "slug": "docsgen",
  "core_files": [
    "docsgen.md",
    "docsgen-00-base-index.md",
    "docsgen-01-base-vision.md",
    "docsgen-02-base-concepts.md",
```

```
    "docsgen-03-base-models.md",
    "docsgen-04-base-phase-types.md",
    "docsgen-05-base-planning.md",
    "docsgen-06-base-execution.md",
    "docsgen-07-base-gate-protocol.md",
    "docsgen-08-base-integrations.md",
    "docsgen-09-base-examples.md",
    "docsgen-10-base-sync.md",
    "docsgen-11-base-schema.md",
    "docsgen-12-arch-index.md",
    "docsgen-13-arch.md",
    "docsgen-14-arch-blazor.md",
    "docsgen-15-arch-data.md",
    "docsgen-16-arch-json.md",
    "docsgen-17-arch-js.md",
    "docsgen-18-arch-visual.md",
    "docsgen-19-arch-updates.md",
    "docsgen-20-arch-macro.md",
    "docsgen-21-arch-validation.md",
    "docsgen-22-arch-performance.md",
    "docsgen-23-arch-testing.md",
    "docsgen-24-arch-deployment.md",
    "docsgen-25-arch-resources.md",
    "docsgen-26-ui-index.md",
    "docsgen-27-ui-spec.md",
    "docsgen-28-ui-uxd.md",
    "docsgen-29-ui-design.md",
    "docsgen-30-ui-layout.md",
    "docsgen-31-ui-navigation.md",
    "docsgen-32-ui-components.md",
    "docsgen-33-ui-interactions.md",
    "docsgen-34-ui-motion.md",
    "docsgen-35-ui-validation.md",
    "docsgen-36-ui-sync.md",
    "docsgen-37-ui-accessibility.md",
    "docsgen-38-ui-examples.md",
    "docsgen-39-appendix-index.md",
    "docsgen-40-appendix-ui.md",
    "docsgen-41-appendix-plugin-structure.md",
    "docsgen-42-appendix-task-reference.md",
    "docsgen-43-diagrams.md"
  ],
  "optional_files": [],
  "excluded_files": [],
  "notes": {
    "link_authority": "All internal links must target files listed in core_files.",
    "numbering_invariant": "Number blocks remain stable across profiles; files may be optional but nu
    "docset_profile": "DocsGen uses the full 43-file structure in strict mode."
  }
}
```

Link Rules (Deterministic)

- Internal links must use **relative paths**.
- A link target must exist in the planned list.

- Links must not point to generation artifacts (planned file list, validation reports) unless the docset profile explicitly includes them.

DocsGen's convention is that core documentation files link only to other core documentation files.

Docset Profiles (Full vs Minimal)

Docset profiles allow the same numbering system to support different deliverables without special-casing prompts.

Full Strict Profile

- All 43 files required
- Diagrams required
- High completeness threshold
- All validators enforced

Lenient Profile

- Diagrams may be optional
- Completeness threshold may be reduced
- Some missing "operational details" can be warnings, but core rules still apply:
  - no placeholders
  - no broken links
  - terminology consistency
  - schema consistency

Regeneration and Planning

Planning is stable across a run.

If a regeneration action updates the planned list, it must be:

- recorded as an explicit artifact change
- validated against existing links
- treated as a controlled migration, not an automatic rewrite

---

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-06-base-execution.md

Base Execution Model

Summary

DocsGen executes as a deterministic state machine.

- Phases are executed sequentially.
- A phase produces artifacts (core docs and supporting reports).
- A deterministic gate validates those artifacts.
- Depending on configuration, the run either:
  - stops for approval (interactive), or
  - continues automatically (batch)

Run Statuses (Canonical)

- **Running** – a phase is currently executing
- **AwaitingApproval** – a phase completed and passed gate validation, but requires user approval
- **Failed** – a gate failed (errors detected) and the run halted
- **Complete** – final validation passed and a bundle was produced
- **Canceled** – the user canceled the run explicitly

Phase Progression Rules

A run may advance from phase N to phase N+1 only if:

1. Phase N artifacts exist
2. Phase N gate validation produced “pass”
3. If review mode is interactive, phase N is approved by a user action

Canonical Execution Loop (Conceptual)

text

```
for phase in [0..5]:
  acquire lock on run
  load inputs and prior approved artifacts
  run agent work (bounded steps)
  write artifacts to store (immutable)
  run deterministic validators
  write validation report artifact
  if validation failed:
     write issues report artifact
     set run status = Failed
     stop
```

```
   if review_mode == interactive:
      set run status = AwaitingApproval
      stop until approved
continue
```

Chunked Generation Pattern

DocsGen avoids “one giant generation” per phase.

Instead:

1. The agent produces a plan (files + outlines).
2. The orchestrator generates file-by-file.
3. Basic checks run after each file (placeholder scan, markdown sanity).
4. The full gate runs after all files are generated.

This pattern is essential because agent runtimes and tool-call runs can expire; smaller steps reduce the blast radius of failure.

Deterministic Gate Outputs

Each gate produces:

- **Human-readable report** (pass summary)
- **Issues report** (only on failure)
- **Structured issue list** (for UI filtering and navigation)

The orchestrator stores these as artifacts and emits run events so that the UI can update in real time.

Partial Regeneration

Partial regeneration is allowed only when explicitly requested.

When re-running a phase:

- earlier successful phases remain immutable
- later phases are not silently rewritten
- only targeted files are regenerated
- all validators for the phase and final cross-validation can be re-run

---

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-07-base-gate-protocol.md

Base Gate Protocol

What a Gate Is

A gate is the deterministic “truth checkpoint” between phases.

A gate exists to answer one question:

> *Are the artifacts produced in this phase safe to treat as stable inputs for later phases?*

If the answer is no, the run stops.

Canonical Gate Steps

1. **Collect Inputs**
   - phase outputs
   - planned file list (link authority)
   - terminology index (if already generated)
   - base schema (if already generated)
2. **Run Validators**
   - placeholder scan
   - internal link resolution
   - schema consistency checks
   - terminology enforcement
   - phase-specific checks (e.g., UI error messages must match canonical rules)
3. **Generate Reports**
   - If passing: `NN-phaseN-validation.md`
   - If failing: `NN-phaseN-issues.md`
4. **Stop or Await Approval**
   - interactive: await approval after pass
   - batch: continue after pass
   - any mode: stop immediately on fail

Pass vs Fail vs Awaiting Approval

Pass

- Zero error-severity issues
- Warnings may exist depending on strictness mode

Fail

- At least one error-severity issue
- Run status becomes Failed
- Orchestrator refuses to overwrite prior artifacts

Awaiting Approval

- Only possible after pass in interactive mode
- Approval is a recorded action (who + when)
- Approval advances the state machine

Strict vs Lenient Behavior

Strictness affects only the severity thresholds, never the core guarantees.

Always Enforced

- no placeholders
- no broken internal links
- terminology consistency
- schema consistency

Strict Only (Typical)

- diagrams required
- completeness threshold higher
- missing operational policies are errors

Lenient (Typical)

- diagrams optional
- completeness threshold lower
- missing operational policies are warnings

Issues Report Format (Human-Readable)

A failing gate produces an issues report that is easy to act on.

Example structure:

md

```
# Phase 3 Issues

## Summary
- 2 broken links
- 1 terminology violation

## Broken Links
- docsgen-12-arch-index.md: link target does not exist: docsgen-24-arch-deployment.md

## Terminology
- docsgen-27-ui-spec.md: uses “pipeline step” but canonical term is “phase”
```

```
## Recommended Fix
- Correct links to planned file targets
- Replace non-canonical terms with canonical terms from Base Concepts

## Next Action
- Fix the affected files manually OR explicitly request regeneration of the affected phase files
```

Immutability Requirement

A gate is also an immutability boundary.

If a core documentation file has been approved in an earlier phase, it must not be overwritten unless:

- the user explicitly triggers a regeneration action, and
- the orchestrator policy allows overwriting in that run mode

---

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-08-base-integrations.md

Base Integrations

Overview

DocsGen integrates four major systems:

1. The host application (RootServer)
2. Azure AI Foundry agents (content generation)
3. Deterministic tools and validators (truth enforcement)
4. Artifact storage and manifests (audit and bundling)

This file describes the integration surfaces conceptually.

Host Integration (RootServer)

DocsGen is designed to run as a plugin inside RootServer’s Blazor application model.

Host responsibilities include:

- providing UI shell and navigation
- providing configuration and secrets (Foundry credentials, storage connection)
- hosting the orchestrator API and event stream

DocsGen does not require a special hosting model beyond standard server-side Blazor + ASP.NET endpoints.

Azure AI Foundry Agents (Intelligence Plane)

Agents are used only to generate Markdown content.

Key integration properties:

- agents are invoked per phase
- phase work is chunked to avoid timeouts
- agent outputs are treated as drafts until gates pass

DocsGen assumes:

- an agent can write content
- deterministic code decides whether the content is acceptable

---

Tool API (Deterministic Surface for Agents)

DocsGen can expose a small OpenAPI tool surface so that agents can:

- read existing artifacts

- write file drafts under orchestrator control
- request deterministic validation execution (optional)

The orchestrator remains authoritative, even if tools are agent-invoked.

Artifact Storage

DocsGen requires an artifact store implementation.

Common options:

- local disk (development)
- blob/object storage (production)

Artifact storage must support:

- content hashing
- immutability guarantees
- listing by run and phase
- bundle creation (zip)

Manifest Storage

A manifest store records:

- runs
- phase executions
- artifact indexes
- validation issues

Manifests enable:

- diffable runs
- audit trails for approvals and overrides
- troubleshooting and reproducibility

Security Model (Conceptual)

DocsGen assumes:

- the UI calls the orchestrator API as an authenticated user
- agents call tool APIs via a service identity (preferred) or restricted token

Key boundary:

- agents must not have the ability to overwrite approved core documentation artifacts
- the orchestrator must enforce this even if an agent attempts it

---

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-09-base-examples.md

Base Examples

Example 1 – Interactive Strict Run (Typical)

Goal: Generate full documentation for a plugin in a reviewable way.

Configuration

json

```
{
  "review_mode": "interactive",
  "validation_strictness": "strict"
}
```

Expected Behavior

- The run stops after each phase.
- A validation report is produced at each gate.
- The user must approve before continuing.
- If a gate fails, an issues report is produced and the run stops.

Typical Output Artifacts

- Phase 1: root + indices + planned file list
- Phase 2: base layer + terminology index
- Phase 3: architecture layer
- Phase 4: UI + appendix
- Phase 5: diagrams + final validation report + bundle zip

Example 2 – Batch Lenient Run (Exploration)

Goal: Generate documentation quickly for early iteration.

Configuration

json

```
{
  "review_mode": "batch",
  "validation_strictness": "lenient"
}
```

Expected Behavior

- The run executes all phases sequentially.
- Validation reports are written but the system does not pause for approval.
- Core guarantees still apply:

- no placeholders
- no broken internal links
- consistent terminology and schema

Example 3 – Partial Regeneration After a Failed Gate

Scenario: Phase 3 fails due to broken links introduced in an architecture file.

Correct Behavior

1. DocsGen produces `03-phase3-issues.md` describing broken links.
2. The run status becomes Failed.
3. Previously approved Phase 1 and Phase 2 artifacts remain immutable.
4. The user either:
   - manually edits the affected architecture files and re-runs validation, or
   - explicitly requests regeneration of Phase 3 only.

Incorrect Behavior (Forbidden)

- silently rewriting Phase 1 or Phase 2 outputs to “fix” Phase 3

Example Issue (Broken Link)

A typical issues report includes actionable locations:

- File: `docsgen-12-arch-index.md`
- Rule: `links.internal.resolves`
- Message: “Link target does not exist: docsgen-24-arch-deployment.md”
- Action: correct the link or regenerate the file

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-10-base-sync.md

Base Synchronization

What “Sync” Means in DocsGen

DocsGen treats documentation as a set of immutable artifacts that can be:

- generated
- validated
- approved
- compared across runs

Synchronization is not collaborative editing; it is the controlled evolution of artifacts across run executions.

Immutability Rules (Canonical)

Core Rule

Once a core documentation file has been produced in a phase that passed validation, the orchestrator must treat it as immutable.

Practical Meaning

- Agents cannot overwrite approved core files.
- A new artifact version must use a new logical path (or explicit version suffix).
- Overwrite attempts are rejected unless a regeneration mode is explicitly enabled.

Versioning Strategy

Artifacts are versioned by:

- run ID
- phase number
- optional execution version

Example logical paths:

text

```
run_2025_12_14_0001/phase2/docsgen-02-base-concepts.md
run_2025_12_14_0001/phase2_v2/docsgen-02-base-concepts.md
```

The manifest records which artifact version is “active” for the run’s latest successful gate.

Regeneration Policy (Canonical)

DocsGen supports three explicit regeneration patterns:

1. **Manual Edit + Revalidate**
   - user edits files externally
   - orchestrator re-runs validators
   - no LLM involved
2. **Regenerate Failed Phase Only**
   - orchestrator loads prior successful artifacts
   - regenerates only targeted phase files
   - re-runs gates
3. **Full Regeneration**
   - user explicitly requests a full rebuild
   - a new run is created, prior run remains archived

Diffing and Auditing

The manifest store enables:

- showing what changed between phase executions
- diffing two runs for the same plugin slug
- answering “why did this gate fail” using structured issue history

---

Conflict Handling

DocsGen expects a single orchestrator authority, but it still must handle:

- two UI clients looking at the same run
- multiple approvals attempted for the same gate
- cancel during generation

The orchestrator resolves these by:

- run-level locks
- idempotent approval actions
- event replay for UI clients

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-11-base-schema.md

Canonical Schema

Purpose

This file defines the canonical JSON structures used by DocsGen to store run state and validation outputs.

These schemas are designed to be:

- deterministic
- easy to validate
- stable across versions

Run Schema

json

```
{
  "runId": "string",
  "pluginSlug": "string",
  "reviewMode": "interactive | batch",
  "validationStrictness": "strict | lenient",
  "currentPhase": "number",
  "status": "running | awaiting_approval | failed | complete | canceled",
  "startedUtc": "string",
  "completedUtc": "string | null",
  "artifactsManifestId": "string"
}
```

PhaseExecution Schema

json

```
{
  "executionId": "string",
  "runId": "string",
  "phaseNumber": "number",
  "agentId": "string",
  "inputs": ["string"],
  "outputs": ["string"],
  "validationResult": "ValidationResult",
  "approval": "Approval"
}
```

Artifact Schema

json

```
{
  "logicalPath": "string",
  "contentHash": "string",
  "sizeBytes": "number",
  "artifactType": "Input | CoreDoc | Report | Plan | Index | Bundle",
  "createdBy": "agent | tool | user",
  "createdUtc": "string"
}
```

ValidationResult Schema

json

```
{
  "status": "pass | fail",
  "summary": "string",
  "issues": ["ValidationIssue"]
}
```

ValidationIssue Schema

json

```
{
  "ruleId": "string",
  "severity": "error | warning",
  "file": "string",
  "message": "string",
  "location": { "line": "number", "column": "number" }
}
```

Approval Schema

json

```
{
  "required": "boolean",
  "status": "pending | approved | rejected",
  "approvedBy": "string | null",
  "approvedUtc": "string | null"
}
```

Planned Files Schema (Generation Artifact)

json

```
{
  "version": "1.0",
  "slug": "docsgen",
  "core_files": ["string"],
  "optional_files": ["string"],
```

```
  "excluded_files": ["string"]
}
```

Terminology Index Schema (Generation Artifact)

json

```
{
  "version": "1.0",
  "slug": "docsgen",
  "terms": [
    {
      "term": "Run",
      "definition": "One end-to-end generation lifecycle.",
      "definedIn": "docsgen-02-base-concepts.md"
    }
  ],
  "forbiddenTerms": [
    {
      "term": "pipeline step",
      "useInstead": "phase",
      "reason": "Phase is the canonical term across the docset."
    }
  ]
}
```

Invariants

- Internal links must point to core documentation files.
- Terminology terms are defined once and used consistently.
- Core documentation artifacts cannot be overwritten after an approved gate.
- Final bundling is permitted only after final validation passes.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-12-arch-index.md

Architecture Layer

This section defines **how DocsGen is implemented and hosted**. It describes the orchestrator, state machine, agent integration, deterministic validators, artifact storage, and the boundary rules that prevent drift and silent regeneration.

- **Architecture Overview**

System decomposition into planes and the high-level responsibilities of each.

- **Architecture – Blazor UI**

Experience plane implementation: pages, components, and UI-to-API contracts.

- **Architecture – Data & Storage**

Artifact store, manifests, hashing, immutability enforcement, and bundle creation.

- **Architecture – JSON Contracts**

Input and output JSON shapes, schema enforcement, and source-of-truth precedence.

- **Architecture – JavaScript**

Optional JS interop and how to keep the system functional without it.

- **Architecture – Visual System**

How progress, issues, and artifacts are visualized (timeline, reports, browsing).

- **Architecture – Updates & Events**

Run event streaming, real-time progress updates, and UI synchronization.

- **Architecture – Macro Execution**

Chunking strategy, partial regeneration, and file-by-file generation loops.

- **Architecture – Validation**

Deterministic validators, gate composition, and structured issue output.

- **Architecture – Performance**

Timeouts, throttling, caching, concurrency, and run budgeting.

- **Architecture – Testing**

Unit tests, integration tests, golden docsets, and regression strategy.

- **Architecture – Deployment**

Hosting inside RootServer, configuration, secrets, and operational controls.

- **Architecture – Resources**

Reference tables: endpoints, configuration keys, rule IDs, and event types.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries

3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-13-arch.md

DocsGen Architecture Overview

Executive Summary

DocsGen is a phase-gated documentation generation system built around a single core idea:

> *The orchestrator is the single source of truth.*

The system is decomposed into four planes to keep responsibilities clean and to prevent correctness from becoming an LLM judgment problem.

Planes (Clean Boundaries)

Experience Plane (Blazor UI)

Owns user interaction:

- input uploads
- run configuration
- phase progress visualization
- approvals
- downloads

Does not call LLMs and does not generate files.

Control Plane (Orchestrator API)

Owns truth and progression:

- run state machine (phases 0–5)
- stop-on-fail behavior
- user gates and approvals
- validator execution
- artifact registry and immutability
- event emission (SignalR/SSE)

Intelligence Plane (Foundry Agents)

Owns drafting:

- generates Markdown content for phase files
- may generate narrative summaries of validator issues

Does not decide pass/fail and must not overwrite approved artifacts.

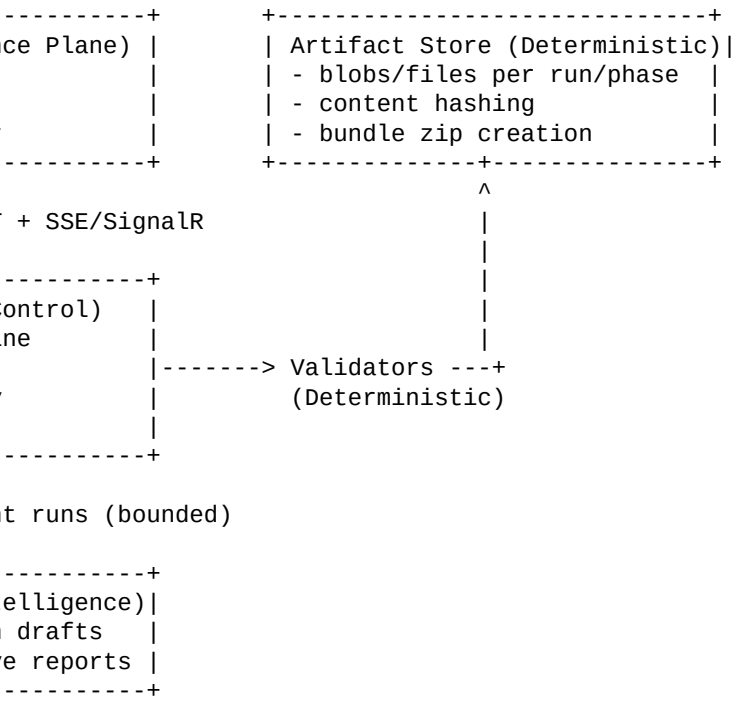
Deterministic Plane (Validators + Storage)

Owns enforcement:

- link checking
- placeholder scanning
- schema and terminology enforcement
- completeness scoring
- artifact persistence and bundling

High-Level Architecture Diagram (Containers)

text



Core Rule: Single Source of Truth

DocsGen forbids “distributed truth.”

- The UI is not authoritative.
- Agents are not authoritative.
- Validators do not advance phases.
- Only the orchestrator can advance the state machine.

This makes run behavior predictable and auditable.

Immutability and Regeneration

Artifacts are immutable by default.

Regeneration must be:

- explicit
- scoped (phase and file targets)
- recorded in the manifest

This is the primary guardrail against drift.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-14-arch-blazor.md

Blazor UI Architecture (Experience Plane)

Overview

DocsGen’s UI is a Blazor experience that exposes the documentation generation workflow without owning any generation logic.

The UI has four jobs:

1. Collect inputs
2. Configure run mode and strictness
3. Display progress and validation results
4. Record approvals and download outputs

All decisions about phases, artifacts, and gate outcomes belong to the orchestrator API.

Page / Screen Set

A minimal DocsGen UI typically includes:

- **New Run Wizard**
  - upload inputs
  - configure review mode and strictness
  - start run
- **Run Dashboard**
  - phase timeline
  - current phase status
  - approve / cancel actions
  - download artifacts (partial or final)
- **Validation Report Viewer**
  - show pass/fail report Markdown
  - show structured issue table (filterable)
- **Artifact Browser**
  - list core files
  - open Markdown preview

Component Responsibilities

RunWizard

- accepts file uploads and configuration
- performs only client-side sanity checks (file present, size)

- calls API to create a run

RunDashboard

- subscribes to run events (SignalR/SSE)
- renders phase status and gate outcomes
- exposes approval and cancellation actions

ValidationViewer

- renders the latest report for a phase
- shows structured issues with “open file” shortcuts

ArtifactBrowser

- lists artifacts grouped by phase
- previews Markdown artifacts
- provides download links for single files or bundles

UI-to-API Contract (Conceptual)

DocsGen’s UI calls a small set of endpoints:

text

```
POST   /api/runs                                  -> create run + upload inputs
GET    /api/runs/run_2025_12_14_0001              -> read run state
GET    /api/runs/run_2025_12_14_0001/events       -> SSE or SignalR stream
POST   /api/runs/run_2025_12_14_0001/approve      -> approve current gate
POST   /api/runs/run_2025_12_14_0001/cancel       -> cancel run
GET    /api/runs/run_2025_12_14_0001/artifacts    -> list artifacts
GET    /api/runs/run_2025_12_14_0001/artifact     -> download single artifact (by path/id)
GET    /api/runs/run_2025_12_14_0001/download     -> download final bundle
```

The UI does not call validator endpoints directly; validation is orchestrator-owned.

Suggested Blazor Patterns (Server-Side)

RootServer is server-side Blazor. DocsGen should follow the same model.

State Handling

- keep view state small and derived from API DTOs
- avoid local “phase state machine” logic
- treat the orchestrator’s `Run.Status` and `Run.CurrentPhase` as authoritative

Streaming Updates

Prefer server push (SignalR/SSE) so progress updates are immediate and do not require aggressive polling.

Example: Run Event Handling (Conceptual)

csharp

```
public record RunEvent(
    string RunId,
    string Type,
    int? Phase,
    string Message,
    DateTimeOffset Utc);

public class RunDashboardState
{
    public string RunId { get; init; } = "";
    public int CurrentPhase { get; set; }
    public string Status { get; set; } = "running";
    public List Events { get; } = new();
}
```

The UI updates purely from events and periodic run snapshots.

UI Guardrails

The UI must not:

- decide whether a gate “should pass”
- generate or rewrite documentation files
- call agents directly
- bypass orchestrator policies

If the UI needs additional actions, the orchestrator API should expose them explicitly.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-15-arch-data.md

Data & Storage Architecture

Overview

DocsGen has two distinct persistence concerns:

1. **Artifact storage** – the immutable files produced by a run
2. **Manifest storage** – the structured index that explains what exists and why

Artifacts are the content. Manifests are the audit trail.

Artifact Storage Requirements

An artifact store must support:

- write artifact (text or binary)
- read artifact
- list artifacts by run and phase
- compute and store a content hash
- enforce immutability rules
- create a bundle zip of core documentation files

Logical Path Convention

text

```
run_2025_12_14_0001/phase3/docsgen-13-arch.md
run_2025_12_14_0001/phase3/03-phase3-validation.md
run_2025_12_14_0001/final/docsgen-documentation-bundle.zip
```

The exact path format can vary, but it must be consistent and stable.

Artifact Store Interface (Conceptual)

csharp

```
public interface IArtifactStore
{
    Task WriteTextAsync(
        string logicalPath,
        string content,
        ArtifactType type,
        string createdBy);

    Task WriteBinaryAsync(
        string logicalPath,
        byte[] content,
        ArtifactType type,
```

```
        string createdBy);

    Task ReadTextAsync(string logicalPath);
    Task ReadBinaryAsync(string logicalPath);

    Task> ListAsync(string runId, int? phase = null);

    Task BundleZipAsync(
        string runId,
        string bundleLogicalPath,
        IReadOnlyList coreFileLogicalPaths);
}
```

Immutability Enforcement

Immutability must be enforced in two layers:

1. **Manifest policy**
   - the orchestrator decides whether an overwrite is allowed
2. **Artifact store guard**
   - the store rejects writes to an immutable path unless an explicit overwrite token is provided

This prevents accidental overwrites when the orchestrator has a bug.

Manifest Storage

The manifest store records:

- Run
- PhaseExecution history
- Artifact index (path, hash, size, type)
- Validation issues (structured)
- Approval events

Why Manifests Matter

Manifests enable:

- showing “what changed” between regenerations
- proving that a bundle matches the validated artifacts
- debugging gate failures without rerunning agents

Structured Issues Storage

Validation issues should be stored as structured objects so the UI can:

- filter by file
- filter by rule
- group by severity
- deep-link into file preview

Bundle Creation

A bundle zip is created only after final validation passes.

Bundle content:

- all 43 core documentation files
- optionally a README pointing to `docsgen.md`

Generation artifacts (validation reports, planned files) are not included in the deliverable bundle.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

File: docsgen-16-arch-json.md

JSON Contracts & Source of Truth

Overview

DocsGen is designed to treat structured JSON as canonical truth, and prose as explanatory.

This file describes the JSON inputs and outputs that form the deterministic backbone of the system.

Primary Input: Plugin Requirements JSON

The plugin requirements document is the highest authority for generation.

It typically contains:

- plugin identity (name, slug, summary)
- entities (schemas, invariants, validation rules)
- user flows (primary flows and roles)
- host integration constraints
- error codes and UI messages
- performance and audit requirements

DocsGen assumes that if the JSON says something, it must be reflected in the documentation without paraphrasing.

Supporting Input: Host Project Structure Reference

DocsGen may ingest:

- a zip of the host project
- a file tree listing
- a curated “structure reference” document

This input is used to:

- map docs to real folders in appendix outputs
- respect host runtime patterns (server-side Blazor, DI conventions)
- avoid documenting forbidden dependencies

Supporting Input: Documentation Formula

DocsGen accepts a documentation formula that defines:

- global rules and constraints
- file templates and required sections

- validation profiles
- regeneration policy

DocsGen treats this as a system configuration input, not a plugin-specific truth source.

Generation Output: Planned File List

The planned file list defines which files exist in the docset and becomes the link authority.

- created before Phase 1 output is generated
- used by link validators in every gate

Generation Output: Terminology Index

DocsGen can extract a terminology index after Base Concepts to enforce consistency across later phases.

This is useful for:

- catching drift (“pipeline step” vs “phase”)
- preventing multiple definitions for the same term

Validation Outputs

Validation outputs exist in two forms:

1. **Human-readable Markdown**
   - pass reports
   - issues reports
2. **Structured issue lists**
   - stored in manifests
   - used by the UI for filtering and deep links

Source-of-Truth Precedence (Enforced)

When inputs conflict:

1. plugin requirements JSON wins
2. host architecture constraints win over example docsets
3. example docsets win only for structure conventions
4. prose instructions are context only

This is a deterministic policy enforced by the orchestrator and validators.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas

2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-17-arch-js.md

JavaScript Architecture (Optional)

Why JavaScript Exists Here

DocsGen does not require JavaScript to function, but optional JS can improve UX:

- richer Markdown preview (syntax highlighting)
- diff viewing (side-by-side or inline)
- large file rendering performance
- drag-and-drop upload UX improvements

DocsGen must remain fully functional without JS enhancements.

Hard Boundary

JavaScript must not:

- call Foundry agents directly
- write artifacts directly to storage
- decide validation outcomes
- advance phases

If JS is used, it must act as a view-layer enhancement only.

Optional JS Modules (Examples)

Markdown Preview Enhancements

- client-side rendering improvements
- code block formatting
- anchor navigation

Diff Viewer

- compare current artifact vs previous artifact version
- highlight changed lines
- expose “download old version” action

Upload UX

- progress indicators for large zip uploads
- chunked uploads (if required by host limits)

JS Interop Contract (Conceptual)

If used, JS should expose small, testable functions:

text

```
docsgen.preview.renderMarkdown(markdownText) -> html
docsgen.diff.compute(oldText, newText) -> diffModel
docsgen.upload.getClientFileStats(file) -> stats
```

All actual data persistence still occurs through the orchestrator API.

Failure Behavior

If JS fails or is unavailable:

- fall back to plain Markdown rendering
- fall back to server-generated diffs (optional)
- keep run control actions fully functional

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-18-arch-visual.md

Visual System Architecture

Overview

DocsGen’s UI is centered around making the run state machine legible.

The UI must make it obvious:

- what phase we are in
- what outputs exist
- why we stopped (pass/fail/awaiting approval)
- what needs human action next

This file describes how those visuals map to deterministic orchestrator state.

Phase Timeline

A timeline is the primary run visualization.

Each phase displays:

- status: not started, running, passed, failed, awaiting approval
- latest report link (validation or issues)
- artifact count and size summary (optional)

Key rule:

- the timeline is derived from orchestrator state and events
- the UI never infers phase success

Validation Report Viewer

The validation viewer renders two complementary views:

1. **Markdown report**
   - pass summary or issues summary
   - human-readable context
2. **Structured issues table**
   - file
   - rule ID
   - severity
   - message
   - jump-to-preview action

This pairing is crucial because Markdown alone is not filterable, and structured issues alone lack narrative.

Artifact Browser

Artifact browsing should support:

- list by phase
- filter by type (core docs vs reports vs inputs)
- preview Markdown artifacts
- download individual artifacts
- download “artifacts so far” zip (optional)

Artifacts shown must match the manifest index.

Approval Gate Visual

If review mode is interactive:

- show a clear “Awaiting Approval” state
- show what passed and what was validated
- show an explicit “Approve & Continue” action

Approval must be recorded through the orchestrator API and reflected in the run history.

Failure Visual

When a phase fails:

- show “Failed” state in timeline
- link to issues report
- show top issues in a summary panel
- offer actions:
  - download artifacts so far
  - regenerate phase (explicit action)
  - cancel run

The UI must not automatically retry or regenerate.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-19-arch-updates.md

Updates & Events (Run Streaming)

Overview

DocsGen exposes run progress via an event stream so the UI can update in real time.

Two common transport options:

- SignalR (native for server-side Blazor)
- Server-Sent Events (SSE)

The transport is not the authority; the orchestrator state is.

Event Design Goals

- Events are append-only and ordered per run.
- Events are sufficient to render progress, but the UI can also fetch snapshots.
- Events include enough data to diagnose what happened without reading logs.

Canonical Event Types

- `run.created`
- `phase.started`
- `artifact.written`
- `validation.started`
- `validation.passed`
- `validation.failed`
- `approval.required`
- `approval.recorded`
- `run.completed`
- `run.failed`
- `run.canceled`

Example Event Payload

json

```
{
  "runId": "run_2025_12_14_0001",
  "type": "validation.failed",
  "phase": 3,
  "message": "Phase 3 gate failed: 2 broken links",
  "utc": "2025-12-14T00:48:10Z",
```

```
  "data": {
    "issuesReportArtifact": "run_2025_12_14_0001/phase3/03-phase3-issues.md",
    "issueCount": 2
  }
}
```

UI Synchronization Strategy

Recommended UI strategy:

1. Subscribe to events
2. Maintain a local event list for the run view
3. Periodically fetch the run snapshot to handle reconnects and missed events
4. Render timeline and reports from authoritative snapshot data

---

Reconnect / Replay

If the UI reconnects:

- it should request events since the last seen timestamp or sequence number
- or fallback to snapshot-only rendering

The orchestrator should support replay by storing events in the manifest store.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-20-arch-macro.md

Macro Execution: Chunking & Partial Regeneration

What “Macro Execution” Means Here

DocsGen uses the term “macro execution” to describe orchestrated work that is larger than a single agent response but still bounded and deterministic.

In practice this means:

- a phase may generate multiple files
- the orchestrator runs generation in a controlled loop
- validators run at the end (and sometimes per file)
- regeneration can target only a subset of files

Phase Execution Pattern (Recommended)

Step 1 – Plan

Ask the phase agent for:

- list of files to generate in this phase
- per-file outline (headings and intent)
- dependencies (which base terms or schemas must be referenced)

Step 2 – Generate File-by-File

For each target file:

- call agent to generate the file content
- write artifact (immutable)
- run immediate checks:
  - placeholder scan
  - markdown structure sanity (non-empty, has H1)
  - navigation block present (if required by convention)

Step 3 – Run Gate Validation

After all files are written:

- run full validators for the phase
- write validation report
- stop or await approval

Why This Pattern Matters

Bounded generation reduces:

- timeout risk
- partial failure blast radius
- drift between files

It also aligns with the regeneration policy:

- regenerate affected files only

Partial Regeneration (File Targets)

DocsGen supports regeneration with explicit targets:

- Phase target: regenerate all files for phase 3
- File target: regenerate only `docsgen-19-arch-updates.md`

Rules:

- prior approved phases are immutable
- regeneration must be explicitly recorded
- revalidation must run after regeneration

Example Regeneration Flow

text

```
Phase 3 fails due to broken links
  |
  v
User requests: regenerate docsgen-12-arch-index.md
  |
  v
Orchestrator:
  - loads planned file list and existing artifacts
  - calls phase agent with specific target file
  - writes new artifact version (phase3_v2)
  - runs Phase 3 validators
  - if pass: await approval (interactive) or continue
```

Approval and Regeneration Interactions

If a phase has already been approved:

- regeneration must create a new execution version
- the UI must show that a newer version exists
- approval must be recorded again for the regenerated phase

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries

3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-21-arch-validation.md

Deterministic Validation Architecture

Overview

DocsGen’s validators are deterministic services that enforce global invariants.

Validators must:

- produce structured issues
- be composable into phase gates
- avoid LLM judgment
- be fast enough to run at every gate

Validator Modules (Canonical)

ForbiddenStringScanner

Detects forbidden placeholder strings and “unfinished” tokens.

Typical forbidden patterns:

- unresolved template tokens
- task markers (unfinished notes)
- ambiguous “fill later” language

MarkdownLinkValidator

Checks internal documentation links.

Rules:

- relative paths only
- targets must exist in the planned file list
- optional: anchor resolution (if anchor linking is used)
- no external links unless explicitly allowed by policy

TerminologyValidator

Enforces canonical terminology as defined by Base Concepts and the terminology index.

Rules:

- defined terms must be used consistently
- forbidden terms must not appear
- acronyms should be defined on first use per file (if policy requires)

SchemaValidator

Validates that:

- the plugin requirements JSON is consistent with schema rules
- entities and fields described in documentation match the JSON truth
- documentation does not introduce new entities

ErrorMessageExactMatchValidator

For UI validation documentation:

- user-facing error messages must match JSON exactly
- no paraphrasing
- error codes must not be orphaned

CompletenessScorer

Computes a coverage score:

- total requirements from JSON
- documented requirements found across the docset
- coverage percentage

Threshold depends on strictness mode.

Gate Composition (By Phase)

Phase 1 Gate

- placeholder scan
- link integrity for indices and root doc
- required sections present in root doc

Phase 2 Gate

- terminology consistency
- schema completeness (entities and invariants)
- link integrity across base files

Phase 3 Gate

- cross-reference to base schema
- no new conceptual terms introduced without definition
- integration coverage documented

Phase 4 Gate

- UI validation strings match canonical rules
- appendix mapping completeness (components/services/models mapped)

Phase 5 Gate

- full docset link integrity
- terminology and schema consistency across all files
- completeness score threshold
- bundle creation allowed only after pass

Structured Issue Output

Validators emit issues with:

- rule ID
- severity
- file
- message
- optional line/column

This enables the UI to provide actionable navigation and filtering.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-22-arch-performance.md

Performance Architecture

Overview

DocsGen performance is defined by three constraints:

1. Agent runtime limits (LLM runs may expire quickly)
2. Validator runtime (must be deterministic and fast)
3. Storage and bundling throughput (must handle many files per run)

DocsGen optimizes for predictability over maximum throughput.

Time Budgeting

Recommended budgeting strategy:

- Phase work is chunked into bounded file generation calls.
- Each file generation call has a strict time budget.
- Validators run in a predictable time window per phase.

Example budgets (guidance):

- per file generation: 30–90 seconds
- per phase validator gate: < 10 seconds for link + placeholder scanning on typical docsets
- final validation: < 60 seconds (full link scan + scoring)

Actual budgets depend on hosting and storage.

Concurrency

DocsGen can support concurrency at two levels:

1. **Across runs**
   - multiple runs can execute concurrently if storage and agent quotas allow
2. **Within a run**
   - validators can run in parallel (safe)
   - generation should remain ordered unless you can guarantee deterministic ordering and stable naming

The orchestrator should always enforce per-run locking so that:

- phase progression is linear
- approvals are idempotent

Caching

Caching is appropriate only for deterministic operations:

- cached reading of large input zips
- cached parsing of planned file list
- cached parsing of headings for anchor validation (if used)

Agent outputs should not be cached across runs unless explicitly enabled, because they are non-deterministic.

Storage Throughput

Bundle creation requires reading all 43 core files and writing a zip.

Recommendations:

- stream reads/writes instead of loading everything into memory
- include a manifest file list in the zip for sanity verification (optional)
- record content hashes in the final validation report

UI Responsiveness

The UI should receive frequent progress events:

- phase started
- file written
- validation started
- validation completed

This avoids the “nothing is happening” effect during long phases.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-23-arch-testing.md

Testing Strategy

Overview

DocsGen must be trustworthy. Testing focuses on determinism, not model quality.

The system should be testable without invoking a real LLM by using:

- stub agents
- golden docsets
- deterministic validator tests

Unit Tests

Validators

- placeholder scanning catches forbidden strings
- link validation resolves all planned file references
- terminology enforcement flags forbidden and undefined terms
- schema checks detect drift between JSON and docs
- completeness scoring is stable

Orchestrator State Machine

- phase progression rules
- stop-on-fail behavior
- approval required behavior in interactive mode
- idempotent approval handling
- cancellation behavior

Integration Tests

Artifact Store

- writes and reads preserve content and hash
- immutability guard prevents overwrite
- list by run/phase returns expected results

API Surface

- run creation with uploads
- event stream returns ordered events
- download bundle returns correct files

Golden Docset Regression

Maintain a golden docset for a known plugin:

- run the validators against it as a regression test
- ensure that validator rule changes are intentional
- diff reports between versions

Golden docsets are a key defense against validator drift.

Agent Testing (Bounded)

Agents are non-deterministic; test them via:

- prompt template tests (static)
- schema-based output validation (deterministic)
- small “smoke runs” in CI (optional)

The orchestrator must remain correct even if agents behave inconsistently.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-24-arch-deployment.md

Deployment & Hosting

Hosting Model

RootServer is a server-side Blazor application with dependency injection and static file hosting.

DocsGen can be hosted in two common ways:

1. **In-Process Plugin**
   - DocsGen UI + orchestrator endpoints live inside the RootServer project
   - simplest for local development
2. **Separated Services**
   - DocsGen UI in RootServer
   - orchestrator API hosted as a sibling ASP.NET service
   - better for scaling and isolating credentials

Both models preserve the same plane boundaries.

Configuration Inputs

DocsGen requires configuration for:

- Foundry agent access (project/endpoint/credentials)
- artifact storage (local path or blob)
- manifest storage (database or file-based store)
- validation profile defaults (strictness, thresholds)

Configuration should be injected via standard RootServer configuration providers.

Secrets Handling

Credentials should never be embedded in documentation files or checked into source control.

Typical secrets:

- Foundry credentials
- storage connection strings
- any tool API authentication material

Operational Controls

Recommended operational controls:

- max concurrent runs

- max upload sizes
- max artifact size per file
- per-run cancellation
- deterministic validator timeouts

Security Boundaries

Key security rules:

- UI actions require authenticated user context
- approval actions must be audited (who + when)
- agent tool calls must be authenticated as a service identity
- agents must not have direct access to overwrite approved artifacts

---

Deployment Outputs

In strict mode, the final output of a successful run includes:

- a final validation report (generation artifact)
- a documentation bundle zip containing 43 core Markdown files

---

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-25-arch-resources.md

Architecture Resources (Reference)

Endpoint Reference (UI-Facing)

text

```
POST   /api/runs
GET    /api/runs/run_2025_12_14_0001
GET    /api/runs/run_2025_12_14_0001/events
POST   /api/runs/run_2025_12_14_0001/approve
POST   /api/runs/run_2025_12_14_0001/cancel
GET    /api/runs/run_2025_12_14_0001/artifacts
GET    /api/runs/run_2025_12_14_0001/artifact
GET    /api/runs/run_2025_12_14_0001/download
```

Tool API Reference (Agent-Facing)

These operations should be small and deliberate:

text

```
ListArtifacts
ReadArtifactText
WriteArtifactText
WriteArtifactBinary
GetPlannedFiles
SetPlannedFiles
ValidateLinks
ValidateNoPlaceholders
ValidateTerminology
ValidateSchemaConsistency
ComputeCompletenessScore
BundleZip
```

Event Type Reference

- run.created
- phase.started
- artifact.written
- validation.started
- validation.passed
- validation.failed
- approval.required
- approval.recorded
- run.completed

- run.failed
- run.canceled

Validator Rule ID Examples

- placeholders.forbidden
- links.internal.resolves
- terminology.forbidden_term
- terminology.undefined_term
- schema.drift
- ui.error*message*mismatch
- completeness.threshold

Phase-to-File Mapping

Phase	Core File Range
Phase 1	docsgen.md, docsgen-00, docsgen-12, docsgen-26, docsgen-39
Phase 2	docsgen-01 through docsgen-11
Phase 3	docsgen-13 through docsgen-25
Phase 4	docsgen-27 through docsgen-38, docsgen-40 through docsgen-42
Phase 5	docsgen-43

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-26-ui-index.md

UI Layer

This section defines how users interact with DocsGen. It documents screens, workflows, interaction rules, and UI validation behaviors.

The UI is designed to be a faithful renderer of orchestrator state.

- **UI Specification**

The user-facing feature set and the primary screens.

- **UI UX Design**

User journeys, mental model, and friction-reduction patterns.

- **UI Visual Design**

Visual language and design rules aligned with the host UI system.

- **UI Layout**

Page structure, layout regions, and responsive behavior.

- **UI Navigation**

Routes and navigation model for runs, artifacts, and reports.

- **UI Components**

Component catalog (wizard, timeline, browser, report viewer, approval controls).

- **UI Interactions**

Exact user interactions and how they map to orchestrator actions.

- **UI Motion**

Motion and transitions used to communicate progress and state changes.

- **UI Validation**

Validation UX, error presentation, and “no paraphrasing” rules.

- **UI Synchronization**

Event stream handling, reconnect behavior, and snapshot fallbacks.

- **UI Accessibility**

Keyboard interaction, screen reader behavior, and accessible patterns.

- **UI Examples**

End-to-end example workflows for common run scenarios.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist

5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-27-ui-spec.md

UI Specification

Overview

DocsGen’s UI is a workflow tool, not a document editor.

It exists to:

- collect inputs
- display run progress
- display deterministic validation results
- record approvals
- provide downloads

The UI does not decide truth; it renders orchestrator truth.

Primary Screens

1) New Run Wizard

Goal: start a run with correct inputs.

Inputs:

- plugin requirements JSON
- host project zip or structure reference
- example docset zip (structure conventions)
- documentation formula
- review mode (interactive/batch)
- validation strictness (strict/lenient)

Outputs:

- creates a run
- shows a run dashboard immediately

---

2) Run Dashboard

Goal: monitor a run and take gate actions.

Must show:

- phase timeline
- current phase status
- latest validation report link
- approval control (interactive mode)

- cancel action
- download artifacts (partial or final)

---

3) Validation Report Viewer

Goal: make failures actionable.

Must show:

- Markdown report (pass report or issues report)
- structured issues table
  - file
  - rule ID
  - severity
  - message
  - jump to preview

---

4) Artifact Browser

Goal: let users inspect generated outputs.

Must support:

- browse by phase
- filter by artifact type
- preview Markdown
- download single artifact

Optional Screens

- Diff viewer (compare artifact versions)
- Settings page (default strictness, storage selection)
- Run history (list runs by plugin slug)

UX Principles

- “Why did we stop?” must be visible without scrolling.
- “What should I do next?” must be explicit.
- The UI must never paraphrase deterministic error messages.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules

4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-28-ui-uxd.md

UI UX Design

Mental Model for Users

DocsGen should feel like a “build pipeline” with checkpoints:

- upload inputs
- run phases
- pass gates
- approve gates (optional)
- download deliverable

Users should never wonder whether a file changed after approval.

Primary User Journey (Interactive Strict)

1. Create run (wizard)
2. Phase 0 preflight passes
3. Phase 1 completes → user reviews indices → approves
4. Phase 2 completes → user reviews base concepts → approves
5. Phase 3 completes → user reviews architecture → approves
6. Phase 4 completes → user reviews UI/appendix → approves
7. Phase 5 completes → final validation → download bundle

At every step, the UI must show:

- latest gate result
- report link
- artifacts produced

Failure Journey

When a gate fails:

- show a failure banner at top of dashboard
- show a short “failure summary” (issue counts by rule type)
- link to issues report
- show structured issues table immediately
- expose next actions:
  - download artifacts so far
  - regenerate phase (explicit)
  - cancel run

Avoid auto-retry or hidden regeneration.

Approval UX (Interactive Mode)

Approval should be explicit and safe:

- show what was validated
- show counts (links ok, placeholders ok, terminology ok)
- show the report before enabling approval
- require a confirmation click (single click is fine; no modal needed if risk is low)

Approval creates an audit event and advances the run.

Batch UX

In batch mode:

- the dashboard should still show phase-by-phase results
- approvals are not requested
- validation failures still stop the run

Batch mode is for speed, not for ignoring correctness.

“Artifacts So Far” UX

Users should be able to download:

- single files
- current phase artifacts
- all artifacts produced so far (optional)

This is valuable even on failure.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-29-ui-design.md

Design System

Tailwind-first approach: describe tokens using Tailwind-like utility mapping.

DocsGen UI is primarily dashboards, lists, and report viewers.

Color Palette

Semantic Status Colors

Status	Suggested Color	Usage
Running	Blue	Active phase, in-progress indicators
Passed	Green	Gate pass badges, success banners
Failed	Red	Gate fail badges, error banners
Awaiting Approval	Amber	Approval-required state, action prompts
Canceled	Gray	Canceled run state

Background & Surface

Token	Guidance	Usage
App background	`bg-zinc-950` or host equivalent	Page background
Surface	`bg-white/5`	Cards, panels
Surface elevated	`bg-white/10`	Hover, selected rows
Border	`border-white/10`	Card and table boundaries

Text

Token	Guidance	Usage
Primary	`text-zinc-100`	Headings and main content
Secondary	`text-zinc-400`	Descriptions, metadata
Muted	`text-zinc-500`	Disabled states, captions

Typography

DocsGen benefits from clear information hierarchy.

Use	Tailwind Guidance

H1	`text-2xl font-semibold`
H2	`text-xl font-semibold`
Section label	`text-sm font-semibold text-zinc-300`
Body	`text-sm leading-6`
Code	`font-mono text-xs leading-6`

Components as Visual Language

Badges

Use small badges for:

- phase number
- status
- strictness and review mode

Tables

Structured issues and artifact lists should be tables with:

- sticky header (optional)
- row hover
- severity column with icon + text (do not rely on color alone)

Markdown Viewer

Markdown previews should use:

- readable line length
- code blocks with scroll
- heading anchors for navigation (optional)

Iconography

RootServer already uses FontAwesome via Blazorise. DocsGen can use icons consistently:

- play/start run
- pause/await approval
- check/pass
- x/fail
- download
- file/text
- warning

Empty States

DocsGen should provide explicit empty states:

- no artifacts yet
- no issues for this phase
- waiting for first event

An empty screen without explanation is treated as a UX bug.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-30-ui-layout.md

UI Layout

Layout Goals

DocsGen’s UI layout must optimize for:

- scanability (phase state is visible immediately)
- actionable failure handling (issues visible without hunting)
- predictable navigation between runs and artifacts

Recommended Dashboard Layout

text

```
+-------------------------------------------------------------+
| Header: DocsGen / Run ID / Status / Mode / Download Button  |
+-----------------------------+-------------------------------+
| Phase Timeline              | Main Panel                    |
| (vertical list)             | - Current phase summary       |
| - Phase 0                   | - Report preview link         |
| - Phase 1                   | - Approval controls (if any)  |
| - Phase 2                   | - Failure banner (if failed)  |
| ...                         |                               |
+-----------------------------+-------------------------------+
| Artifact Browser (optional docked region)                   |
+-------------------------------------------------------------+
```

Validation Viewer Layout

text

```
+-------------------------------------------------------------+
| Header: Phase / Status / Back to Dashboard                  |
+-----------------------------+-------------------------------+
| Markdown Report (left)      | Structured Issues Table (right)|
| - narrative summary         | - file, rule, severity, msg   |
| - sections                  | - filter + search             |
+-----------------------------+-------------------------------+
```

This dual-pane approach supports both “read the summary” and “fix the issues.”

Artifact Browser Layout

Artifact browsing benefits from grouping:

- by phase

- by artifact type
- by last modified time

Provide a preview panel for Markdown and a download button per artifact.

Responsive Behavior

On narrow viewports:

- timeline collapses to a horizontal stepper
- validation viewer becomes stacked (report then issues table)
- artifact browser becomes a separate route or drawer

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-31-ui-navigation.md

Navigation & Information Architecture

Route Map (Suggested)

DocsGen navigation is centered on runs.

text

```
/docsgen
/docsgen/runs
/docsgen/runs/new
/docsgen/runs/run_2025_12_14_0001
/docsgen/runs/run_2025_12_14_0001/phase/3
/docsgen/runs/run_2025_12_14_0001/artifacts
/docsgen/runs/run_2025_12_14_0001/artifact?path=run_2025_12_14_0001/phase3/docsgen-13-arch.md
```

The exact route prefix can match the host's conventions.

Breadcrumb Model

Breadcrumbs should reflect the user's location:

- DocsGen
- Runs
- Run ID
- Phase / Artifact (optional)

Example breadcrumb:

text

```
DocsGen › Runs › run_2025_12_14_0001 › Phase 3
```

Navigation Principles

- The run dashboard is the "home" for a run.
- Reports and artifacts are reachable without losing the run context.
- Deep links must work (open a report directly via URL).

Global Actions Placement

Recommended global actions:

- **New Run** (primary action on /docsgen/runs)
- **Download Bundle** (primary action on a completed run dashboard)

- **Cancel Run** (secondary action, but visible on running runs)
- **Approve & Continue** (primary action when awaiting approval)

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-32-ui-components.md

Component Library

1) Run Wizard Components

RunWizard (Page / View)

Purpose: Collect inputs and configuration to create a run.

Sections

- Plugin requirements upload
- Host project reference upload
- Example docset upload
- Documentation formula upload
- Mode selection (review mode, strictness)
- Start button

Primary States

State	Visual	Behavior
Empty	guidance text + disabled Start	Start disabled until required inputs present
Ready	Start enabled	POST to create run
Uploading	progress indicator	prevent duplicate submits
Error	error banner	show API error details

FileDropZone

Purpose: Drag/drop + browse upload control.

Rules

- Must show accepted file types
- Must show file size and name after selection
- Must support replace/remove

Suggested UI

text

```
+------------------------------+
| Drop file here or Browse     |
| - shows file name + size     |
| - Replace / Remove actions   |
+------------------------------+
```

---

2) Run Dashboard Components

RunHeader

Purpose: Make the run state obvious.

Content:

- run ID
- plugin slug
- status badge
- mode badges
- key actions (download, cancel)

PhaseTimeline

Purpose: Visualize phase progression.

Timeline Row Content

- phase number + label
- status icon
- report link (if exists)
- issue counts (if failed)

Phase Row States

Phase State	Icon	Meaning
Not Started	circle	phase has not run yet
Running	spinner	generation or validation in progress
Passed	check	gate passed
Awaiting Approval	pause	gate passed but approval required
Failed	x	gate failed, run halted

ApprovalPanel

Purpose: Provide safe continuation in interactive mode.

Content

- “Awaiting Approval” message
- link to validation report
- “Approve & Continue” button
- approval audit metadata once approved

EventLogPanel (Optional)

Purpose: Show ordered run events.

Should support:

- newest-first toggle
- filter by phase
- copy event payload

---

3) Validation Components

ValidationReportViewer

Purpose: Render the Markdown report for a phase.

Rules

- display report exactly as stored
- do not paraphrase or reword

IssuesTable

Purpose: Make structured issues actionable.

Columns:

- severity
- rule ID
- file
- message
- location (line/col)
- open preview action

Filters:

- severity filter
- file filter
- rule filter
- search by message text

---

4) Artifact Components

ArtifactBrowser

Purpose: Explore artifacts produced by the run.

Grouping:

- by phase
- by type

Actions:

- preview
- download

MarkdownPreview

Purpose: View Markdown artifacts.

Features

- renders headings clearly
- supports code block scrolling
- optional table styling

DownloadPanel

Purpose: Provide output downloads.

Should support:

- “Download bundle” when complete
- “Download artifacts so far” on failure (optional)
- per-file download

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-33-ui-interactions.md

UI Interactions

Principle

Every interaction in the UI maps to an explicit orchestrator action. The UI does not “guess” the next phase or decide validity.

Start Run

User Action

Click “Start Run” in the Run Wizard.

UI Behavior

- validate required files are selected
- POST to create run
- navigate to run dashboard

Orchestrator Behavior

- stores input artifacts
- runs Preflight
- emits run events

Approve & Continue (Interactive Mode)

User Action

Click “Approve & Continue” when the run is awaiting approval.

UI Behavior

- call approval endpoint
- disable the button until response
- update timeline on events

Orchestrator Behavior

- records approval (who + when)
- advances to next phase
- emits approval recorded event and next phase started event

Cancel Run

User Action

Click “Cancel Run”.

UI Behavior

- confirm intent (inline prompt or modal)
- call cancel endpoint
- lock UI controls afterward

Orchestrator Behavior

- stops execution loop
- emits run canceled event
- preserves artifacts produced so far

View Validation Report

User Action

Click a report link on the timeline.

UI Behavior

- navigate to report viewer route
- fetch report Markdown artifact
- fetch structured issue list for the phase

Download Bundle

User Action

Click “Download bundle” on a completed run.

UI Behavior

- request download
- show file name and size

Orchestrator Behavior

- ensures final validation passed
- returns bundle artifact

Regenerate Phase (Explicit)

Regeneration must be explicit; if supported in the UI:

User Action

Click “Regenerate Phase 3” (only available when failed or explicitly enabled).

UI Behavior

- show explanation: regeneration creates a new execution version
- call orchestrator regenerate endpoint (if implemented)

Orchestrator Behavior

- creates a new phase execution version
- regenerates targeted files only
- reruns gate validation
- emits events and returns to awaiting approval (interactive)

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-34-ui-motion.md

Animation & Motion

Timeline Motion

- **Phase status change**: icon cross-fade (150–200ms)
- **Active phase highlight**: subtle pulse or background tint
- **Gate pass**: check icon pop-in (150ms)
- **Gate fail**: brief shake on failure banner (avoid excessive motion)

Motion exists to communicate state changes, not to decorate.

Report Viewer Motion

- **Navigate to report**: content fade-in (150ms)
- **Issue row selection**: highlight transition (100ms)

Toast Notifications (Optional)

Use toasts for:

- “Run created”
- “Approval recorded”
- “Download started”

Avoid using toasts for validation failures; failures should be persistent and visible until resolved.

Accessibility Rule

All motion should respect reduced-motion settings. If reduced motion is enabled:

- disable shakes and pulses
- use instant state changes

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

File: docsgen-35-ui-validation.md

UI Validation

Principle: No Paraphrasing

DocsGen’s UI must never paraphrase deterministic validation output.

If a validator emits:

- rule ID
- message
- location

The UI displays those fields as-is.

This prevents “helpful rewriting” from causing confusion or hiding the real rule that failed.

Validation Presentation Model

Each phase presents validation in two layers:

1. **Report Markdown**
   - summary and narrative context
2. **Structured Issues**
   - actionable rows for fix/regeneration decisions

Severity Handling

Error

- gate fails
- run stops
- UI shows failure banner and issues table

Warning

- gate may pass depending on strictness profile
- UI should still show warnings (collapsed by default is acceptable)

Common Validation UI Patterns

Failure Banner

A persistent banner at the top of the dashboard should include:

- phase number

- short summary (counts)
- link to issues report

Issue Table Filters

Support filtering by:

- severity
- file
- rule ID

Copy Rules

The UI may add a very small amount of additional copy such as:

- “This run is awaiting approval.”
- “Phase 3 gate failed.”

But it must not rewrite validator messages.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-36-ui-sync.md

UI Synchronization

Overview

DocsGen UI synchronization is event-driven.

The UI should subscribe to a run’s event stream and also be able to fetch a snapshot if:

- it reconnects
- it missed events
- it is opened via a deep link

Recommended Strategy

1. Subscribe to events for the run
2. Append events to an in-memory list for display
3. When a “phase completed” or “validation completed” event arrives, fetch the latest snapshot for correctness
4. On reconnect, fetch snapshot first, then request recent events (if supported)

Polling Fallback

If streaming is unavailable:

- poll the run snapshot endpoint on a modest interval
- keep polling slower during idle states (awaiting approval)
- keep polling faster during running phases

Polling must not become the primary architecture unless required by the host.

Idempotency

UI actions must be idempotent-safe:

- approval should handle double-click without advancing twice
- cancellation should handle repeated clicks without error
- downloads should be repeatable

The orchestrator is responsible for idempotency, but the UI should also disable buttons while calls are in flight.

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-37-ui-accessibility.md

Accessibility

Accessibility Goals

DocsGen UI should be usable with:

- keyboard-only navigation
- screen readers
- reduced motion preferences
- high contrast needs

Accessibility is not optional; it is required for review workflows.

Keyboard Navigation

- All interactive controls must be reachable by tab.
- Timeline rows should be keyboard navigable.
- Issue table rows should support keyboard selection and “open preview” action.

Screen Reader Support

- Status badges must have text labels, not color-only meaning.
- Phase timeline should expose phase number and state in accessible text.
- Validation issues should be readable as a table with proper headers.

Color and Contrast

- Do not rely only on color for status (use icons and text labels).
- Ensure contrast on banners (failed vs warning vs success).
- Provide a consistent focus ring style.

Markdown Preview Accessibility

- Headings should render as semantic headings.
- Code blocks should be scrollable without trapping focus.
- Links must be keyboard accessible.

Reduced Motion

If reduced motion is enabled:

- disable pulses and shakes
- avoid animated spinners if possible (replace with “Running” text)

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

■ Examples (Concrete UI Scenarios)

Example A – Starting a Run

1. User opens `New Run`
2. Uploads plugin requirements JSON, host zip, example docset zip, and documentation formula
3. Selects:
   - Review mode: interactive
   - Strictness: strict
4. Clicks “Start Run”
5. UI navigates to run dashboard and shows Phase 0 running

Example B – Approval Gate

1. Phase 1 completes and passes validation
2. Timeline shows Phase 1: “Awaiting Approval”
3. Approval panel shows:
   - link to `01-phase1-validation.md`
   - “Approve & Continue”
4. User clicks approve
5. Timeline updates to Phase 2 running

Example C – Gate Failure

1. Phase 3 fails validation
2. UI shows failure banner:
   - “Phase 3 gate failed: 2 broken links”
3. Timeline marks Phase 3 as failed
4. Clicking the report opens issues viewer
5. Issues table shows rows with:
   - file
   - rule ID
   - message
   - line number
6. User downloads “artifacts so far” and fixes the file externally

---

Example D – Download Final Bundle

1. Phase 5 completes and passes final validation
2. Run status becomes Complete
3. Download panel shows “Download bundle”
4. User downloads a zip containing exactly 43 Markdown files

Example E – Regenerate a Failed Phase (Explicit)

1. Phase 4 fails due to a UI validation mismatch
2. User clicks “Regenerate Phase 4” (explicit action)
3. UI shows a warning:
   - regeneration creates a new phase execution version
4. Orchestrator regenerates only Phase 4 files
5. Gate runs again and the UI returns to awaiting approval

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-39-appendix-index.md

Appendix Index

Extra information about DocsGen.

- **UI Technical Notes** – UI implementation reference and state patterns
- **Plugin Structure Mapping** – how DocsGen files fit into the RootServer plugin structure
- **Task Reference** – implementation roadmap mapped to documentation sections

---

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-40-appendix-ui.md

Appendix – UI Technical Notes

This appendix captures implementation-focused UI notes that are useful during build-out.

State Management (Guidance)

DocsGen UI state should be derived from orchestrator DTOs.

Recommended approach:

- keep a `RunSnapshot` DTO model
- keep a list of `RunEvent` entries for the current view
- recompute timeline rows from snapshot + latest event data

Avoid:

- duplicating the orchestrator's state machine in the UI

Markdown Rendering

Two acceptable strategies:

1. Server-side rendering of Markdown into HTML
2. Client-side rendering using a small JS helper

Rules:

- never rewrite report content
- display content exactly as stored in artifacts

Large File Handling

Artifact previews should:

- stream content where possible
- avoid loading very large files into the DOM all at once
- provide "download instead" for oversized files

Component Testability

Components should be testable by:

- injecting a fake API client
- feeding static snapshots and events
- verifying correct rendering of statuses and issues

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-41-appendix-plugin-structure.md

Appendix – Plugin Folder Structure Mapping (Blazor / .NET)

This appendix explains how a DocsGen plugin can be placed into the **standard plugin folder structure** used by RootServer (similar to existing plugins).

The goal is to keep responsibilities clean and predictable:

- **00_Core** = foundations and contracts
- **Displays / Fields** = UI primitives
- **Pointers / Actions** = navigation + commands
- **Components / Widgets / View / Panels** = composition layers

Static assets (JS/CSS/images) belong in:

- `wwwroot/` (hosted by RootServer)

Standard Folder Responsibilities (RootServer Convention)

`00_Core/`

Foundational code used everywhere.

- records/models (Run, Artifact, Issue, Policy)
- enums (status, phase numbers)
- interfaces (artifact store, validators, API clients)
- serialization helpers and DTOs

`01_Displays/`

View-only UI (no user input).

- status badges
- read-only phase row renderer
- artifact metadata displays

`02_Fields/`

Interactive UI inputs.

- upload controls
- strictness / mode selectors
- filters for issues table

`03_Pointers/`

Navigation and selection primitives.

- run picker
- breadcrumb controls

- deep-link helpers

`04_Actions/`

Commands and side effects.

- start run action
- approve action
- cancel action
- download action
- any orchestration API client calls

`05_Nodes/` **(Optional for DocsGen)**

DocsGen is not a graph domain, but keeping the folder can preserve convention.

Suggested use:

- domain-specific UI models for run timeline nodes (pure view models)
- helper mappers between DTOs and UI presentation

`06_Components/`

Composable UI building blocks.

- RunWizard form component
- PhaseTimeline component
- IssuesTable component
- ArtifactBrowser component
- ReportViewer component

`07_Widgets/`

Medium/large layout compositions.

- RunDashboard widget (timeline + summary + actions)
- ValidationViewer widget (report + issues table)

`08_View/`

Full screen views.

- `V_DocsGenRuns.razor`
- `V_DocsGenRunDashboard.razor`
- `V_DocsGenReportViewer.razor`

`09_Panels/`

Host-level containers.

- optional if DocsGen lives inside a broader editor shell

Suggested DocsGen Plugin Tree

text

```
RootServer/Shared/Plugins/DocsGen
■■■ 00_Core
■   ■■■ Models
■   ■   ■■■ Run.cs
■   ■   ■■■ PhaseExecution.cs
■   ■   ■■■ ArtifactInfo.cs
■   ■   ■■■ ValidationIssue.cs
■   ■   ■■■ ValidationResult.cs
■   ■■■ Policies
■   ■   ■■■ ValidationProfile.cs
■   ■   ■■■ RegenerationPolicy.cs
■   ■■■ Services
■   ■   ■■■ IDocsGenApiClient.cs
■   ■   ■■■ DocsGenApiClient.cs
■   ■   ■■■ IRunEventsClient.cs
■   ■   ■■■ RunEventsClient.cs
■   ■■■ Validation
■       ■■■ IValidator.cs
■       ■■■ LinkValidator.cs
■       ■■■ PlaceholderValidator.cs
■       ■■■ TerminologyValidator.cs
■■■ 01_Displays
■   ■■■ D_StatusBadge.razor
■   ■■■ D_PhaseBadge.razor
■■■ 02_Fields
■   ■■■ F_FileUpload.razor
■   ■■■ F_StrictnessSelect.razor
■   ■■■ F_ReviewModeSelect.razor
■■■ 03_Pointers
■   ■■■ P_RunBreadcrumbs.razor
■   ■■■ P_RunPicker.razor
■■■ 04_Actions
■   ■■■ A_StartRun.razor
■   ■■■ A_ApproveRun.razor
■   ■■■ A_CancelRun.razor
■   ■■■ A_DownloadBundle.razor
■■■ 06_Components
■   ■■■ C_RunWizard.razor
■   ■■■ C_PhaseTimeline.razor
■   ■■■ C_ArtifactBrowser.razor
■   ■■■ C_ReportViewer.razor
■   ■■■ C_IssuesTable.razor
■■■ 07_Widgets
■   ■■■ W_RunDashboard.razor
■   ■■■ W_ValidationViewer.razor
■■■ 08_View
    ■■■ V_DocsGenRuns.razor
    ■■■ V_DocsGenRunDashboard.razor
    ■■■ V_DocsGenReportViewer.razor
```

Documentation Placement (RootServer Docs)

To match existing patterns, store the Markdown docset under:

text

```
RootServer/Docs/Extend/Module-Internal/DocsGen/
```

This mirrors the NodeEditor documentation placement and keeps internal module docs grouped consistently.

Mapping Documentation to Implementation

This section links key docs to the code you typically create.

Orchestrator + Planes

- Architecture overview: **docsgen-13-arch.md**
- Validation architecture: **docsgen-21-arch-validation.md**
- Storage architecture: **docsgen-15-arch-data.md**

UI Workflows

- UI specification: **docsgen-27-ui-spec.md**
- UI components: **docsgen-32-ui-components.md**

Operational Controls

- Deployment: **docsgen-24-arch-deployment.md**
- Task reference: **docsgen-42-appendix-task-reference.md**

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

File: docsgen-42-appendix-task-reference.md

Appendix – Implementation Checklist & Documentation Map

This appendix provides a practical roadmap for building DocsGen. Each implementation milestone references the documentation that should guide it.

This checklist is a guide, not a strict order.

Milestone 1: The Spine (Phase 0 + Phase 1)

Goal: A run can be created, preflight can pass/fail deterministically, and foundation files exist.

- **Run model + state machine skeleton**

→ **Base Execution Model** → **Base Models**

- **Artifact store abstraction + hashing**

→ **Architecture – Data & Storage** → **Base Synchronization**

- **Phase 0 preflight checks**

→ **Base Phase Types** → **JSON Contracts**

- **Phase 1 outputs (root + indices + planned files)**

→ **Base Planning** → **Base Gate Protocol**

Milestone 2: Deterministic Validators (Phase Gates)

Goal: validators run reliably and stop-on-fail is enforced.

- **Placeholder scanning**

→ **Architecture – Validation** → **UI Validation**

- **Internal link validation against planned file list**

→ **Base Planning** → **Architecture – Validation**

- **Terminology index extraction and enforcement**

→ **Base Concepts** → **Architecture – Validation**

- **Completeness scoring**

→ **Architecture – Validation** → **Canonical Schema**

Milestone 3: Foundry Agent Integration (Phase 2–4 Generation)

Goal: agents generate Markdown under orchestrator control without owning truth.

- **Agent runner and bounded file generation loop**

→ **Macro Execution** → **Base Execution Model**

- **Tool API surface (optional)**

→ **Architecture Resources** → **Base Integrations**

Milestone 4: UI (Wizard + Dashboard + Viewer)

Goal: the UI becomes usable end-to-end.

- **New Run Wizard**

→ **UI Specification** → **UI Components**

- **Run Dashboard and timeline**

→ **Visual System** → **UI Layout**

- **Report viewer with issues table**

→ **UI Validation** → **Updates & Events**

Milestone 5: Final Validation + Bundle Output (Phase 5)

Goal: full cross-validation passes and a bundle is produced.

- **Final docset validation**

→ **Architecture – Validation** → **Diagrams**

- **Zip bundling and download**

→ **Architecture – Data & Storage** → **Deployment**

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---

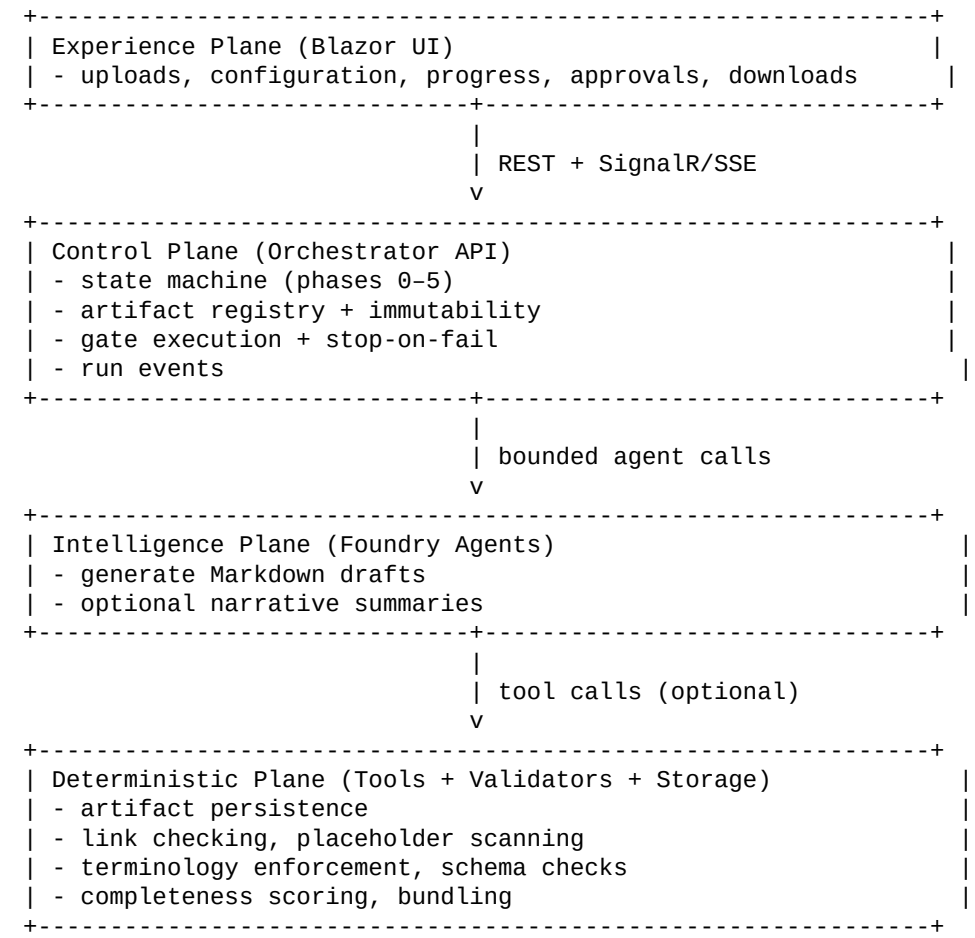
File: docsgen-43-diagrams.md

Diagrams & Flows

This file contains diagrams for the DocsGen system.

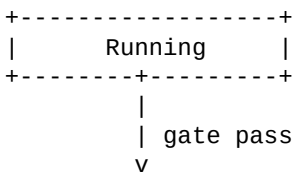
1) Four-Plane System (Boundaries)

text



2) Orchestrator State Machine (Run Status)

text



```
                  +------------------+
                  | AwaitingApproval |  (interactive only)
                  +--------+---------+
                           |
                           | approval
                           v
                  +------------------+
                  |      Running     |
                  +------------------+

Gate fail at any time:
    Running -> Failed

Cancel at any time:
    Running/AwaitingApproval -> Canceled

Final pass:
    Running -> Complete
```

---

3) Phase Progression (0–5)

text

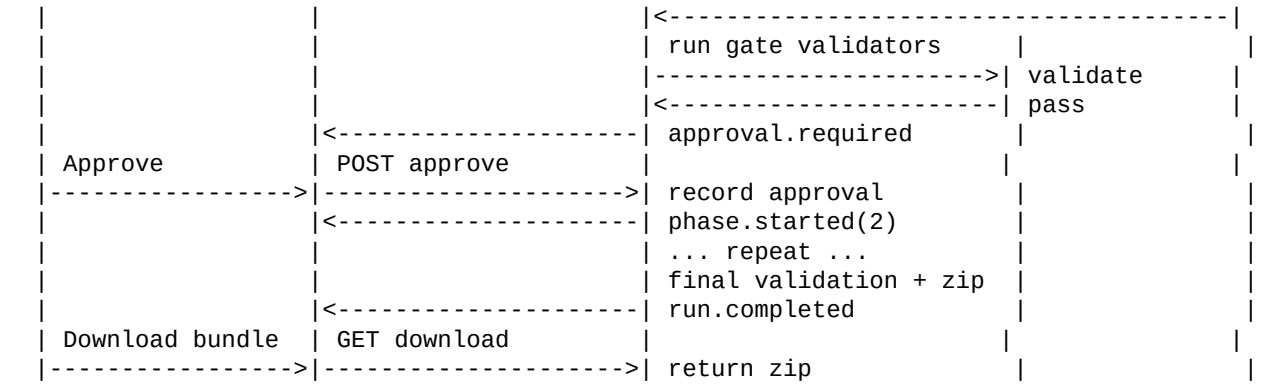
```
Phase 0  Preflight (blocking)
  |
  v
Phase 1  Foundation (root + indices + planned files)
  |
  v
Phase 2  Base Layer (concepts + models + schema)
  |
  v
Phase 3  Architecture Layer (implementation + integration)
  |
  v
Phase 4  UI + Appendix (workflows + mapping)
  |
  v
Phase 5  Diagrams + Cross-Validation + Bundle
```

---

4) Sequence – Interactive Strict Run

text

```
User               UI                Orchestrator            Validators         Agents
  |                 |                      |                      |              |
  | Upload inputs   |                      |                      |              |
  |---------------->| POST /api/runs       |                      |              |
  |                 |--------------------->| create run           |              |
  |                 |                      | preflight checks     |              |
  |                 |                      |--------------------->| run preflight |
  |                 |                      |<---------------------| pass/fail     |
  |                 |  events              |                      |              |
  |                 |<---------------------| phase.started(1)      |              |
  |                 |                      | generate Phase 1 files |              |
  |                 |                      |--------------------------------------->|
```



---

5) Sequence – Gate Failure and Stop-on-Fail

text

```
Orchestrator              Validators
     |                        |
     | run gate validation    |
     |----------------------->|
     |<-----------------------| fail (errors)
     |
     | write issues report
     | set run status = Failed
     | emit validation.failed event
     v
UI shows failure banner + issues table
(no further phases run)
```

---

6) Sequence – Partial Regeneration (Explicit)

text

```
User/UI                    Orchestrator                    Agents                Validators
  |                            |                            |                      |
  | Click "Regenerate Phase 3" |                            |                      |
  |--------------------------->| create phase3_v2 execution  |                      |
  |                            | generate targeted files     |                      |
  |                            |--------------------------> |                      |
  |                            |<---------------------------- |                      |
  |                            | run Phase 3 validators      |--------------------->|
  |                            |                            |<---------------------|
  |                            | if pass -> awaiting approval / continue            |
```

---

7) Artifact Path Layout

text

```
run_2025_12_14_0001/
■■■ input/
■   ■■■ plugin-requirements.json
■   ■■■ host-project.zip
```

```
■■■ phase1/
■   ■■■ docsgen.md
■   ■■■ docsgen-00-base-index.md
■   ■■■ docsgen-12-arch-index.md
■   ■■■ docsgen-26-ui-index.md
■   ■■■ docsgen-39-appendix-index.md
■   ■■■ 01-planned-files.json
■■■ phase2/
■   ■■■ docsgen-01-base-vision.md
■   ■■■ ...
■   ■■■ 02-terminology-index.json
■■■ phase3/
■   ■■■ docsgen-13-arch.md ... docsgen-25-arch-resources.md
■■■ phase4/
■   ■■■ docsgen-27-ui-spec.md ... docsgen-42-appendix-task-reference.md
■■■ phase5/
■   ■■■ docsgen-43-diagrams.md
■   ■■■ 05-final-validation.md
■   ■■■ docsgen-documentation-bundle.zip
```

---

8) Validator Pipeline (Conceptual)

text

```
Artifacts Produced
   |
   v
Placeholder Scan  -----> Issues (structured)
   |
   v
Link Validation   -----> Issues (structured)
   |
   v
Terminology Check -----> Issues (structured)
   |
   v
Schema Consistency ----> Issues (structured)
   |
   v
Completeness Score ----> Score + threshold decision
   |
   v
Gate Pass/Fail
```

---

9) RootServer Plugin Placement (Suggested)

text

```
RootServer/
■■■ Shared/
■   ■■■ _Editor/
■   ■■■ Plugins/
■       ■■■ Airtable/
■       ■■■ DocsGen/
■           ■■■ 00_Core/
■           ■■■ 01_Displays/
```

```
■           ■■■ 02_Fields/
■           ■■■ 03_Pointers/
■           ■■■ 04_Actions/
■           ■■■ 06_Components/
■           ■■■ 07_Widgets/
■           ■■■ 08_View/
■■■ Docs/
    ■■■ Extend/
        ■■■ Module-Internal/
            ■■■ DocsGen/
                ■■■ (43 markdown files)
```

← DocsGen

1. **Base** – conceptual foundations, terminology, and canonical schemas
2. **Architecture** – orchestration, agents, validators, storage, and hosting boundaries
3. **UI** – user workflows, screens, components, and interaction rules
4. **Appendix** – folder mapping, task references, and implementation checklist
5. **Diagrams** – architecture, state machines, and end-to-end sequences

---