ChatGPT

Automated Code Documentation Tools for Security-Focused Python Projects: Comprehensive Research Report

Executive Summary

Bottom Line: For your security-focused LLM orchestration platform requiring formal accreditation, use **Sphinx + sphinx-autodoc-typehints** as your primary documentation framework, supplemented by **pyreverse** for architecture visualization, **pydeps** for dependency analysis, and **Bandit/Semgrep** for security control documentation. This combination provides the rigor, extensibility, and industry acceptance needed for security accreditation while maintaining developer productivity. Additionally, plan for **formal methods** (e.g. TLA+ or Alloy specifications) in the design of critical components *from the outset* to catch subtle design bugs early and avoid last-minute reengineering [1].

Top 3 Recommended Tool Combinations

1. Core Documentation: Sphinx with Security Extensions ☆☆☆☆☆
- **Why**: Sphinx is the industry standard for Python project docs (used by Django, OpenStack, Flask) [2] [3]. It supports the best cross-referencing for complex security hierarchies and can output multiple formats including HTML and **PDF** (a compliance must-have) [2]. Extensive extensions (autodoc, Napoleon, MyST) allow custom security directives and integration with tooling.
- **Effort**: 2-3 days initial setup, 1-2 hours/week maintenance
- **Cost**: Free and open-source

2. Architecture Visualization: pyreverse + pydeps + PyCG
- **Why**: **pyreverse** (from pylint) generates UML class and package diagrams to illustrate system structure and trust boundaries. **pydeps** maps module dependencies and highlights circular imports (critical for identifying unintended tight coupling or privilege escalation paths). **PyCG** provides call graphs for dynamic analysis, aiding threat modeling of data flows. Together, they give a clear picture of system architecture and potential attack surfaces.
- **Effort**: 1-2 days integration (automate diagram generation in CI), minimal ongoing effort
- **Cost**: Free and open-source

3. Security Analysis: Bandit + Semgrep + Safety
- **Why**: **Bandit** scans Python code for security issues (with CWE identifier mapping for industry-standard weakness tracking [4]). **Semgrep** enables custom rule sets for detecting project-specific anti-patterns or verifying that security controls (like auth checks) are present. **Safety** (or pip-audit) checks dependencies for known vulnerabilities (CVEs). This trio covers code, configuration, and dependency risks, providing continuous feedback on security posture.
- **Effort**: 1 day integration, then fully automated in CI
- **Cost**: Free tiers available (open-source); enterprise options for advanced reporting

Timeline and Effort Estimates

Phase 1: Foundation (Weeks 1-2)
- Sphinx setup with basic project structure and **sphinx-autodoc-typehints**: *2 days*
- Initial autodoc configuration and docstring formatting: *1 day*
- CI/CD documentation pipeline (GitHub Actions workflow for docs): *1 day*
- Security documentation outline and templates (e.g. placeholder for threat model, controls): *2 days*
- **(Include formal methods planning)** Identify any critical algorithms/protocols for which formal specification might be beneficial; set up environment or training for TLA+/Alloy if needed: *1 day*

Phase 2: Automation (Weeks 3-4)
- Integrate **pyreverse/pydeps** for automated diagram generation (UML class diagrams, dependency graphs): *2 days*
- Set up **Bandit/Semgrep** in CI for continuous security scanning (fail builds on high-severity findings): *1 day*
- Automatic diagram generation script (using Graphviz/PlantUML) triggered in CI: *1 day*
- Pre-commit hooks for doc linting (doc8) and security checks (Bandit) to prevent drift: *1 day*

Phase 3: Security Documentation & Verification (Weeks 5-8)
- Document **trust boundaries** and component responsibilities (manual write-up using architecture diagrams): *1 week*
- Create **data flow diagrams (DFDs)** for key use cases (user auth, plugin execution, etc.): *1 week*
- Perform **threat modeling** (STRIDE or similar) on major components; document threats and mitigations: *1 week*
- Compile a **security control inventory** (list of implemented security controls mapped to requirements): *1 week*
- **Formal modeling of critical logic** (if applicable): Develop a formal specification for one critical module (e.g., the plugin sandbox or authentication flow) and validate key properties using a tool like TLA+: *1 week (overlaps with other tasks)*. *This ensures early detection of design flaws and provides high-assurance evidence.*

Phase 4: Accreditation Preparation (Weeks 9-12)
- Map implementation to **NIST 800-53 controls** (20 families) and other framework requirements (FedRAMP, CMMC): *2 weeks*. Produce a control matrix with references to documentation, code, tests, and formal proofs (if any).
- Draft the **System Security Plan (SSP)** and other required documentation (privacy impact assessment, contingency plan, etc.), pulling from the content in Sphinx: *1 week*
- **Evidence collection and automation**: Assemble outputs from CI (scan reports, test results, formal verification results) as accreditation evidence artifacts: *1 week*
- Conduct an internal **pre-assessment audit** using the documentation and address any gaps: *1 week*

Total Initial Investment: ~12 weeks (3 months) of focused effort for setup and documentation. Ongoing maintenance will require ~4-6 hours per week to update docs, review security scans, and keep diagrams in sync with code changes.

Key Risks and Mitigations

1. **Documentation Drift:** As code evolves, docs can become outdated. *Mitigation:* Use CI “doc check” gates (fail build if documentation is not updated alongside code), and schedule monthly doc audits. Leverage Sphinx autodoc to regenerate API docs on each build to catch changes, and include doc updates in the definition of done for each ticket.

2. **Plugin Architecture Complexity:** The plugin system is dynamic, which makes documenting all interactions challenging. *Mitigation:* Clearly document the plugin lifecycle and security model (capabilities, namespace restrictions) in a dedicated section. Use runtime introspection to generate tables of available plugin hooks or permissions. Regularly review plugin-related docs with dev team to ensure accuracy.
3. **Learning Curve for Sphinx:** Developers may not be familiar with reStructuredText or Sphinx configurations. *Mitigation:* Provide team training (a two-day workshop and reference cheat sheets). Start with a simple structure and gradually enhance it. Use MyST Markdown to allow writing docs in Markdown if that is more comfortable.
4. **Accreditation Requirements Change:** Compliance standards (FedRAMP, CMMC, etc.) might update their documentation expectations. *Mitigation:* Design the documentation strategy to be modular. For example, if the framework changes, update a mapping file or section without rewriting everything. Keep an eye on policy updates and allocate time in each release cycle to adjust documentation for compliance.
5. **Over-Automation Gaps:** Relying too much on generated documentation might miss contextual details (why certain security decisions were made). *Mitigation:* Maintain a healthy balance (~70% automated, 30% manual). For each major feature or component, write a short narrative explaining its security posture, assumptions, and decisions. Enforce human review for all generated content (no blind faith in tools).
6. **Late Introduction of Formal Methods:** If formal verification is pushed off or done at the very end, it could reveal fundamental design issues when it's expensive to fix. *Mitigation:* Identify early which components (e.g. authentication flow, encryption module, consensus algorithm) warrant formal specification. Train one or two team members in a specification language (TLA+ recommended) by mid-project, and pilot a formal model on a critical piece by Phase 3. This way, any insights can be folded into the design when changes are still manageable. *Industry experience shows formal methods can uncover bugs that other techniques miss [1], making early adoption worthwhile.*

---

1. DETAILED TOOL COMPARISON MATRIX

API Documentation Generators

Tool	Setup	Security Features	Type Hints Support	Cross-Refs	PDF Output	CI/CD Integration	Best For
Sphinx	☆☆☆ Medium-High	☆☆☆☆☆ Extensible (many plugins for security content)	☆☆☆☆☆ Full support	☆☆☆☆☆ Best-in-class links	Yes (latexpdf)	☆☆☆☆ Mature (ReadTheDocs friendly)	Enterprise-grade security projects (e.g. Django, OpenStack) [2] [3]
MkDocs	☆☆ Low	☆☆☆☆ via plugins (MkDocs has a OWASP plugin)	☆☆☆☆☆ Excellent (via mkdocstrings)	☆☆☆☆ Good (less automatic)	⚠ Plugin needed	☆☆☆☆☆ Easy (Material theme CI)	Modern clean API sites, lower complexity projects

Tool	Setup	Security Features	Type Hints Support	Cross-Refs	PDF Output	CI/CD Integration	Best For
pdoc	☆ Very Low	☆☆ Basic (limited to docstrings)	☆☆☆☆ Good	☆☆☆ Limited cross-module	No	☆☆☆☆ Simple (one-command)	Quick internal tools, smaller codebases
pydoctor	☆☆ Low-Med	☆☆ Limited (Twisted-specific)	☆☆☆☆ Good (supports typing)	☆☆☆☆ Good (for its scope)	No	☆☆☆ CI friendly	Niche projects (Twisted/Zope), API ref generation
Doxygen	☆☆☆☆ High	☆☆ Some (via add-ons, not Python-specific)	☆☆ Partial	☆☆☆ Moderate	Yes	☆☆☆ Available	Mixed-language projects (C++/Python combos)

Winner: Sphinx – It offers the most flexibility and power for security-focused documentation. Key advantages: - Custom **directives** and domains for security content (you can create a `.. threat::` directive, for example). - **Type hint integration** out-of-the-box (via autodoc and Napoleon) to ensure consistent documentation of interfaces. - **Cross-referencing** across the entire project (classes, methods, glossary terms, even between the code and manual sections) is unparalleled, which is crucial for tracing requirements to implementation. - **PDF output** for formal deliverables (e.g., for auditors who require a PDF export) is built-in [2]. - Proven in large projects (the Python standard library, Django, OpenStack all use Sphinx) [2] [3], meaning there is community knowledge and many existing extensions. - Hosted docs are easy via Read the Docs or GitHub Pages, including versioning which is helpful for accreditation (e.g., freeze docs for each release or audit).

Runner-up is **MkDocs** with Material theme, which provides a great UI and easy setup (popular for many modern projects). However, MkDocs lacks built-in PDF generation and has slightly weaker cross-referencing for large codebases. It can be made to work (e.g., the OWASP MkDocs PDF plugin), but Sphinx edges it out for our needs.

Architecture Visualization Tools

Tool	Diagram Types	Output Formats	Accuracy	Scalability	Security Relevance	Best For
pyreverse	UML class diagrams, package dependency graphs	PNG, SVG, PlantUML, DOT	☆☆☆☆ (uses AST, fairly accurate class models)	☆☆☆☆☆ (handles large codebases by focusing on high-level relations)	☆☆☆ Shows class inheritance and composition (useful for trust boundaries)	Visualizing class structure and module relationships
pydeps	Module import dependency graph (static)	SVG, PNG	☆☆☆☆☆ (truthful to imports)	☆☆☆☆ (can get busy on big projects, but has filtering)	☆☆☆☆ Highlights tight coupling and potential layering violations (security risk if critical logic spans modules)	Analyzing and breaking down dependencies, spotting cycles
code2flow	High-level call graphs (simplified)	PNG, SVG, PDF	☆☆☆ (heuristic, may miss dynamic calls)	☆☆ (not great for very large projects)	☆☆ Basic call flows, not security-specific	Small scripts or simplified flow charts
py2puml	UML class diagrams from code (via PlantUML)	PlantUML text	☆☆☆☆ (leverages AST like pyreverse)	☆☆☆ (suitable for mid-size code)	☆☆ Focus on class UML only (less on security)	Codebase where PlantUML is preferred in pipeline
pyan3	Static call graph analysis (functions)	PNG, SVG	☆☆☆ (limited by static analysis of Python)	☆☆ (works for smaller projects)	☆☆ Can find potential recursion or unexpected couplings	Simple call graphs, academic use

Tool	Diagram Types	Output Formats	Accuracy	Scalability	Security Relevance	Best For
PyCG	Python Call Graph (advanced static analysis)	JSON output (post-process for graph)	☆☆☆☆☆ (uses bytecode analysis, more complete)	☆☆☆☆☆ (scales, but output needs filtering)	☆☆☆☆☆ Great for security because it can map taint flows or privilege changes through calls	Deep call graph analysis and integration with security tooling (e.g., to find if unsafe functions are reachable)

Recommended Combination: Utilize **multiple tools in tandem** to cover different views: - **pyreverse** for structural diagrams: Generate class diagrams of the core system and the security-related modules (authentication, plugin sandbox, etc.). This will visually communicate inheritance relationships, composition, and key interfaces. Mark these diagrams with notes about trust boundaries (e.g., which classes run in untrusted plugin context vs. core context).
- **pydeps** for dependency graphs: Run pydeps on the entire project (with clustering by package) to illustrate how components depend on each other. Focus on any **circular dependencies** or unexpected dependencies (for example, if a low-level security utility ends up depending on a high-level module, that might violate design intent). Pydeps can show if your security layer is cleanly separated.
- **PyCG** for dynamic call graph analysis: Especially on critical paths like authorization checks, use PyCG to generate a call graph. This can help ensure that, for instance, all code paths that mutate sensitive data go through an authorization check function. You can post-process the JSON to visualize or to query (e.g., ensure that `check_permission` is in the call graph of all functions modifying protected resources).

All these can be automated. For example, you might include a script to generate these diagrams and save them in `docs/diagrams/generated/` whenever the code changes significantly. This ensures your documentation always reflects the actual code structure.

Static Analysis and Security Scanning Tools

Tool	Language/ Scope	Security Focus	Output Formats	CI/CD Friendly	Doc Integration	False Positives
Bandit	Python (code)	☆☆☆☆☆ Checks for common Python security issues (bad functions, injections, etc.) [4]	JSON, CSV, HTML, SARIF	☆☆☆☆☆ (designed for CI)	☆☆☆☆ (can embed results in docs as HTML report)	Low (mature ruleset)

Tool	Language/ Scope	Security Focus	Output Formats	CI/CD Friendly	Doc Integration	False Positives
Semgrep	Any (configurable)	☆☆☆☆☆ Pattern-based rules (OWASP, custom rules for app logic)	JSON, SARIF, text	☆☆☆☆☆ (very CI friendly)	☆☆☆☆ (can link rule IDs to docs)	Low (rules can be tuned)
Safety	Python (packages)	☆☆☆☆ Dependency vulnerability (CVE) scanning	Text, JSON	☆☆☆☆☆ (pip-audit similar)	☆☆☆ (list of vulns can be referenced)	Very Low (database-driven)
Pyre	Python (code)	☆☆☆ Taint analysis, type safety (security-oriented checks for data flow)	JSON	☆☆☆☆ (requires setup)	☆☆ (mostly separate)	Medium (some tuning needed)
mypy	Python (code)	☆☆ Type checking (catches type errors, some are security relevant)	Text, JSON (via plugins)	☆☆☆☆	☆☆ (enforce type hints in docs)	Low (if types well-annotated)
pip-audit	Python (packages)	☆☆☆☆ Similar to Safety (CVE checks)	Text, JSON	☆☆☆☆☆	☆☆☆ (list of vulns)	Very Low

Recommended Stack: Leverage complementary tools to cover different angles: 1. **Bandit** – The go-to Python SAST (Static Application Security Testing) tool. It detects issues like use of `eval`, weak cryptography, hard-coded passwords, Django SQL injection potential, etc. Bandit maps findings to CWE identifiers where possible [4], which helps when doing compliance mapping (you can say which CWE each issue relates to, aligning with OWASP Top 10 or NIST 800-53 requirements). Integrate Bandit into pre-commit and CI (fail the build on any **High** severity finding). 2. **Semgrep** – Write custom rules to enforce project-specific security requirements. For example, if all plugins must call a certain validation function, write a Semgrep rule to ensure `validate_data(data)` is present in any plugin `execute()` method. Use the rich community rulesets for common frameworks (Flask, FastAPI, Django) to catch misconfigurations or missing security headers. Run Semgrep in CI with a results summary in the docs (perhaps include the Semgrep report JSON/HTML in an appendix). 3. **Safety/pip-audit** – Automate the checking of third-party packages for known vulnerabilities. This is critical since using a vulnerable version of a library could compromise your system. Run weekly (or on every dependency change) and feed results to the security documentation (maybe a section “Dependency Security Status” that lists current packages and any known issues). 4. **Additional**: Use **mypy** to enforce type hints (not directly a security tool, but it can prevent certain classes of errors and makes the autodoc cleaner) and consider **Pyre** or **CodeQL** for deeper security analysis if the project grows in complexity (e.g., needing data-flow analysis).

By combining these, you establish a robust *automated security guardrail* that not only finds issues but also produces artifacts (reports, logs) that can be cited in your accreditation evidence. Each tool's output can be archived (e.g., store Bandit's JSON report from the release build as evidence that you have an automated SAST process per compliance requirements).

---

2. IMPLEMENTATION GUIDE

This section provides step-by-step guidance to implement the recommended toolchain and integrate it into your project. It includes project structure recommendations, configuration snippets, and automation scripts. All documentation content for the security accreditation should reside in a dedicated subfolder of your documentation directory – this keeps it separate from the user-facing "how to run the system" docs. For example, if `docs/` is currently used for user guides, create `docs/security-architecture/` for this detailed security documentation.

Step 1: Sphinx Setup for Security Documentation

Installation: Start by adding Sphinx and relevant extensions to your project. Create a `docs/requirements.txt` for documentation-specific Python packages. For example:

```
pip install sphinx sphinx-rtd-theme sphinx-autodoc-typehints \
            sphinx-pyreverse myst-parser sphinxcontrib-mermaid
```

(We include sphinx-pyreverse for directly embedding UML diagrams, and myst-parser to allow Markdown files like ADRs to be included.)

Documentation Structure: In the repository, set up a docs subdirectory for the security documentation. One approach is to have `docs/` for general docs and a subfolder like `docs/architecture/` for this security-focused documentation. For instance:

```
your-project/
├── docs/
│   ├── user-guide/              # existing user-facing docs (if any)
│   │   └── index.rst
│   ├── architecture/            # security & architecture docs (our focus)
│   │   ├── conf.py              # Sphinx config for architecture docs
│   │   ├── index.rst            # master doc for architecture/security
documentation
│   │   ├── security/
│   │   │   ├── architecture.rst
│   │   │   ├── trust-boundaries.rst
│   │   │   ├── data-flows.rst
│   │   │   ├── threat-model.rst
│   │   │   ├── controls.rst
│   │   │   └── compliance/      # e.g., compliance mappings
│   │   │       ├── nist-800-53.rst
│   │   │       ├── fedramp.rst
│   │   │       └── cmmc.rst
│   │   ├── api/
│   │   │   ├── core.rst          # API docs (autogenerated)
│   │   │   ├── plugins.rst
│   │   │   ├── datasources.rst
│   │   │   └── tools.rst
│   │   ├── adr/
│   │   │   ├── index.rst         # list of Architectural Decision Records
│   │   │   ├── 001-tech-stack.md
│   │   │   ├── 007-plugin-isolation.md
│   │   │   └── template.md
│   │   ├── diagrams/
│   │   │   ├── generated/        # UML and graph outputs (added via script,
gitignored)
│   │   │   └── static/           # any hand-drawn or static images
│   │   └── _static/
│   │       ├── bandit-report.json
│   │       ├── bandit-report.html
│   │       └── semgrep-report.json
│   └── requirements.txt          # sphinx and extensions requirements
├── src/elspeth/                  # your source code
│   ├── core/
│   ├── plugins/
│   ├── datasources/
│   └── tools/
└── .github/workflows/
    └── docs.yml                  # CI workflow for docs
```

Sphinx Configuration: Below is a snippet for `docs/architecture/conf.py` focusing on security documentation needs:

```
project = 'Elspeth – Secure LLM Orchestration Platform'
author = 'Security Team'
release = '1.0'

extensions = [
    'sphinx.ext.autodoc',          # standard API doc
    'sphinx.ext.autosummary',
    'sphinx.ext.napoleon',         # Google style docstrings
    'sphinx.ext.viewcode',         # link to source code
    'sphinx.ext.intersphinx',      # references to external docs if needed
    'sphinx_autodoc_typehints',    # better type hints display
    'myst_parser',               # to include Markdown (for ADRs)
    'sphinxcontrib.mermaid',       # Mermaid diagrams (for DFDs if desired)
    'sphinx_pyreverse',            # custom extension we will create or
configure to run pyreverse
]
```

```
# Autodoc settings
autodoc_default_options = {
    'members': True,
    'undoc-members': False,
    'show-inheritance': True,
    'special-members': '__init__',
    'inherited-members': True,
}
autodoc_typehints = "description"  # put type hints in the function
description
autodoc_inherit_docstrings = True

# Napoleon settings (Google/NumPy docstring support)
napoleon_google_docstring = True
napoleon_numpy_docstring = False

# HTML output
html_theme = 'sphinx_rtd_theme'
html_title = "Elspeth Security Documentation"
html_logo = "_static/your_logo.png"  # if you have a logo
html_favicon = "_static/favicon.ico"

# LaTeX/PDF output (for formal deliverables)
latex_documents = [
    ('index', 'Elspeth-Security.tex', 'Elspeth Security Documentation',
     'Elspeth Security Team', 'manual'),
]
```

Key points: We include `sphinxcontrib.mermaid` for diagrams (Mermaid can be used for quick DFDs or flow charts in the docs), and a hypothetical `sphinx_pyreverse` extension. Since sphinx-pyreverse isn't an official extension, you might write a small extension that calls pyreverse and includes the generated diagrams, or simply generate diagrams as image files and reference them in the RST (e.g., using the `.. image::` directive). The conf is set to produce a LaTeX/PDF (`latex_documents`) which is useful for compliance submissions.

At this stage, also create placeholders for various manual docs: - `architecture.rst`: high-level architecture description. - `trust-boundaries.rst`: describe the zones (e.g., plugin sandbox vs core, network segments, etc.). - `data-flows.rst`: narrate key data flow diagrams (user request, plugin execution, etc.). - `threat-model.rst`: STRIDE or similar analysis. - `controls.rst`: list of security controls and how they are implemented in the system. - `compliance/nist-800-53.rst`, etc.: a mapping of each control to implementation evidence.

Using a toctree, include these in `index.rst` (the main file) so they appear in order. For example, in `docs/architecture/index.rst`:

```
.. toctree::
   :maxdepth: 2
   :caption: Contents:
```

```
   security/architecture
   security/trust-boundaries
   security/data-flows
   security/threat-model
   security/controls
   security/compliance/nist-800-53
   adr/index
   api/core
   api/plugins
   api/datasources
   api/tools
```

This will ensure the documentation is organized and navigable.

Step 2: Automated Diagram Generation

To keep architecture diagrams up-to-date, automate their creation.

Diagram Generation Script: Create a script (e.g., `docs/architecture/generate_diagrams.sh`) that invokes pyreverse, pydeps, etc.:

```bash
#!/bin/bash
set -e

# Directories
OUTPUT_DIR="docs/architecture/diagrams/generated"
SRC_DIR="src/elspeth"

mkdir -p "${OUTPUT_DIR}"

echo "Generating UML diagrams with pyreverse..."
# Full system class diagram (may be large, perhaps limit to core for clarity)
pyreverse -o png -p Elspeth "${SRC_DIR}" -o "${OUTPUT_DIR}"
# You can also generate package diagrams for specific subpackages if needed
pyreverse -o png -p ElspethCore "${SRC_DIR}/core" -o "${OUTPUT_DIR}"

echo "Generating dependency graph with pydeps..."
pydeps "${SRC_DIR}" --max-bacon=2 --output "${OUTPUT_DIR}/dependencies.svg"
--show-cycles

# If needed, generate call graph for a critical component (like plugin
system)
echo "Generating call graph for plugin manager..."
pycg --package elspeth --fast "${SRC_DIR}/core/plugin_manager.py" > "$
{OUTPUT_DIR}/plugin_manager_callgraph.json"
# (Later, this JSON can be converted to a diagram or used in analysis.)
```

Include this script in your repo, and ensure that the CI can run it (you may need Graphviz and Java or PlantUML if using those output formats). This script will output files like `classes_Espeth.png` (UML class diagram) and `dependencies.svg`. You then reference these in the documentation. For example, in `architecture.rst` you might include:

```
.. image:: ../diagrams/generated/classes_Elspeth.png
   :alt: UML class diagram of Elspeth core components

.. image:: ../diagrams/generated/dependencies.svg
   :alt: Module dependency graph (highlighting module interactions and any
cycles)
```

This way, whenever the code changes and the script is re-run, the images update. (Ensure the images folder is `.gitignore`-d if you prefer not to store diagrams in VCS, or commit them for traceability—a decision to note in an ADR.)

Step 3: CI/CD Integration for Docs and Security Scans

Set up continuous integration to automatically build docs, run security scans, and push documentation outputs (HTML or PDF) for review.

GitHub Actions Workflow: (`.github/workflows/docs.yml`)

```
name: Documentation and Security CI

on:
  push:
    branches: [ main ]
  pull_request:
    branches: [ main ]

jobs:
  docs:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: '3.11'

      - name: Install dependencies
        run: |
          sudo apt-get update && sudo apt-get install -y graphviz
          pip install -r docs/requirements.txt
          # Install project in case autodoc needs it
          pip install -e .
      - name: Generate diagrams
        run: bash docs/architecture/generate_diagrams.sh

      - name: Run Security Scans
        run: |
          bandit -r src/ -f html -o docs/architecture/_static/bandit-
report.html
          bandit -r src/ -f json -o docs/architecture/_static/bandit-
report.json
          semgrep --config auto --json src/ > docs/architecture/_static/
semgrep-report.json
          # Optionally, run Safety for dependencies
          safety check -r requirements.txt -r requirements-prod.txt > docs/
architecture/_static/deps_vuln.txt || true

      - name: Build HTML Docs
        run: |
          cd docs/architecture
          make html  # assuming Makefile is set up in docs/architecture for
Sphinx
          make latexpdf  # build PDF as well

      - name: Upload artifact (HTML documentation)
        if: failure() && github.event_name == 'pull_request'
        uses: actions/upload-artifact@v3
        with:
          name: docs-html
          path: docs/architecture/build/html

      - name: Deploy to GitHub Pages
        if: github.ref == 'refs/heads/main' && success()
        uses: peaceiris/actions-gh-pages@v3
        with:
          publish_dir: docs/architecture/build/html
          github_token: ${{ secrets.GITHUB_TOKEN }}
          # You might host this on a private site or internal server if
sensitive
```

This workflow does a few important things: - Installs system deps (Graphviz for diagrams, etc.). - Runs the `generate_diagrams.sh` to produce up-to-date diagrams. - Runs Bandit and Semgrep, outputting their results into the docs static folder. (The HTML Bandit report could be embedded via an iframe or linked, and the JSON can be parsed to produce summary tables if desired.) - Builds the Sphinx docs (HTML and PDF). - On PRs, it uploads the docs as an artifact (so reviewers can download and see changes easily). - On main branch, it publishes to GitHub Pages. (For a private project, you might skip pages and instead distribute the PDF or restrict access.)

Pre-Commit Hooks: Also set up a pre-commit configuration to catch issues early:

`.pre-commit-config.yaml`:

```
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.5.0
    hooks:
      - id: check-yaml
      - id: end-of-file-fixer
      - id: trailing-whitespace

  - repo: https://github.com/PyCQA/bandit
    rev: 1.7.5
    hooks:
      - id: bandit
        args: ['-f', 'short', '-r', 'src']

  - repo: https://github.com/pycqa/doc8
    rev: 0.11.2
    hooks:
      - id: doc8
        args: ['--max-line-length=120', '--ignore-path', 'docs/architecture/
_build']

  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.4.1
    hooks:
      - id: mypy
```

This will run Bandit and doc8 on each commit. The developer gets immediate feedback if they introduce a high-severity security issue or break doc style guidelines.

Formal Methods Integration (if chosen): If you decide to proceed with formal specs for a component, consider adding a job in CI to run model checks. For instance, if using TLA+, you might have TLA+ specs in `docs/architecture/formal/` and use the `tlc` model checker in CI to ensure the spec satisfies invariants. While optional, automating this ensures your formal specs are consistently checked as code changes (especially if you link spec parameters to code constants, etc.).

Step 4: Writing and Linking Documentation Content

With the skeleton in place, start filling in the docs. Some key guidelines while writing:

- **Use consistent terminology** – e.g., if the system is divided into *Core* vs *Plugin Sandbox*, define these terms in a glossary or upfront in `architecture.rst`. Then use them precisely in the threat model, boundaries, etc.
- **Cross-reference heavily** – Sphinx allows references like `:ref:` and `:doc:` and `:meth:`. For example, in the threat model discussion of a plugin vulnerability, you can link to the API docs of the `PluginManager.verify_signature` method to show where it's implemented. In code docstrings, refer to manual docs: in a class docstring, you might include "See :doc:`/security/threat-model` for the threat analysis of this component."
- **Keep manual sections (30%) focused** – They provide narrative, rationale, and context that code docs cannot. Use them for overviews, reasoning, and compliance mappings. Leave lower-level

details (like parameter descriptions) to autodoc unless a particular parameter has security impact.

For example, document an important class with links:

```
class PluginManager:
    """Manages plugin lifecycle and enforces security isolation.

    **Security Architecture**: This component is responsible for loading
plugins in an isolated environment.
    It implements multiple controls (signature verification, capability
tokens, resource limits).
    See :doc:`/security/architecture` and :doc:`/security/trust-boundaries`
for how plugins are isolated.

    **Threat Model**: Relevant threats include plugin impersonation and
sandbox escape.
    These are analyzed in :doc:`/security/threat-model` (see "Plugin System
Threats").

    **Related Design Decisions**: Architectural decisions regarding plugin
isolation are recorded in :doc:`/adr/007-plugin-isolation`.

    **Compliance**: Implements controls for NIST 800-53: SC-3 (Security
Function Isolation), and SC-39 (Process Isolation).
    """
    # class implementation...
```

Conversely, in the manual `architecture.rst`, you might include:

```
The **PluginManager** in the core is the gatekeeper for all plugin
interactions. It isolates plugins by:

- Restricting accessible built-ins and imports (only safe modules).
- Running all plugin code under a unprivileged OS-level role (no admin
rights).
- Performing digital signature verification on plugin packages before
loading.

.. note:: This approach is further detailed in the code documentation
of :py:class:`elspeth.core.plugin_manager.PluginManager`.
```

This cross-linking ensures that someone reading the docs can jump to code, and someone reading code can jump to docs, which auditors love because it shows nothing is a black box.

Step 5: Formal Methods and Design Assurance (Optional but Recommended)

While not every project includes formal specifications, for high-security contexts it can be a powerful addition. Identify if any part of your system warrants formal modeling. Candidates: - Complex security-critical state machines (authentication/token expiry, role-based access control logic). - Concurrency or race-condition sensitive logic (if any, e.g., revocation, transaction commit). - Cryptographic protocol or data confidentiality logic.

If you proceed: - Choose a tool like **TLA+** for system design properties or **Alloy** for data model constraints. - Write the spec concurrently with development of that component (e.g., as you build the plugin sandbox, specify in TLA+ the allowed and disallowed plugin actions). - Verify key properties (e.g., "a plugin cannot modify core data without going through PluginManager", or "auth tokens cannot be forged"). - Document this in a `formal-spec.rst` or within the relevant section. For example, in the threat model, include a note:

```
**Formal Verification:** The isolation property is formally specified and
verified using TLA+.
We model the core and plugin as separate processes and show that any plugin
action that changes core state must have a corresponding permission from the
core (see formal model file in `docs/architecture/formal/
plugin_isolation.tla`).
This gives high confidence that a compromised plugin cannot escape its
sandbox under the specified assumptions.
```

- If possible, include the TLA+ (or other spec) as an appendix or link to it in your repository. (Even if auditors don't understand it deeply, it demonstrates a rigorous approach.)

By considering formal methods early, the design of the system can still be influenced by what you learn (for example, if the formal model shows a potential race condition in plugin unloading, you can adjust the design now rather than after deployment).

---

3. SECURITY DOCUMENTATION STRATEGY

A successful security documentation approach balances **automation** and **manual insight**. Approximately **70%** of the content can be generated or derived from code and tools, while **30%** should be expert-written analysis, rationale, and contextual documentation.

Automated Documentation (≈ 70%)

These sections will be kept in sync with the code and system state through automation:

- **API Reference** – Generated by Sphinx autodoc from docstrings. Every public class, function, and module in the platform should be covered. This ensures that *what the code does* is transparently documented. Developers must be disciplined to write comprehensive docstrings with security notes as needed (e.g., note if a function assumes input is already validated).
- **Security Control Implementation Mapping** – Using structured data (maybe a YAML or CSV), you can generate a table or matrix of security controls. For example, maintain a file `docs/architecture/control_matrix.csv` mapping control IDs to implementation references (code or config). A short Python script can convert that to RST tables. This way, if a control implementation changes, updating the CSV keeps docs accurate.
- **Diagrams and Architectural Views** – As discussed, auto-generate class diagrams, dependency graphs, and even call graphs. When integrated into the docs, this provides an always up-to-date

architectural view. If the architecture changes (new module, changed dependency), the next docs build will reflect it.

- **Security Scan Reports** – Instead of writing "we run Bandit and it finds no issues" manually (which becomes stale immediately), include actual data. For example, parse the Bandit JSON and list the count of issues by severity, or include a snippet of the Bandit HTML report. This serves as evidence of your secure coding practices. Similarly, include a summary of Safety checks (e.g., "No known vulnerable packages as of last scan on 2025-10-10").
- **Test and Coverage Stats** – While not strictly documentation, including a high-level summary of security-related test coverage can be persuasive. For instance, "All cryptographic functions have 100% branch coverage by tests; overall test coverage is 92%." If you use a coverage tool, you can automatically embed the coverage percentage badge or a table of coverage by module.
- **Changelogs/Release Notes** – Keep a changelog that highlights security-relevant changes (e.g., "v1.2: Improved input validation on API X (addresses potential injection risk)"). This can be part of docs and largely autogenerated from commit messages or PR notes (using tools like towncrier).

By automating these, you reduce manual effort and ensure the documentation reflects the current state of the system and processes.

Manual Documentation (≈ 30%)

Critical areas that require human insight, justification, and explanation:

1. **Security Design Rationale** – Explain *why* certain decisions were made. For example, why you chose a particular encryption scheme, or why the plugin system is implemented in-process versus out-of-process. This often takes the form of an Architectural Decision Record (ADR) or a section in the docs. It should reference threat models, requirements, and alternatives considered. For instance:

```
Design Rationale – Plugin Isolation
-------------------------------
**Decision:** Implement plugin sandboxing via restricted namespaces and
capabilities within the main process (instead of separate microservices
for each plugin).

**Reasoning:** Running plugins in-process avoids inter-process
communication overhead and complexity, which is acceptable given our
threat model assumes plugins are vetted and signed. We mitigate the in-
process risk with strict runtime restrictions:
(a) disabling dangerous built-ins, (b) limiting memory and CPU, (c)
validating all plugin outputs.

**Alternatives Considered:** Out-of-process plugins (rejected due to
performance and much higher implementation complexity), WebAssembly
sandbox (rejected due to immature Python WASM support at the time).

**Consequences:** This approach is efficient but means a malicious
plugin could attempt to interfere with the host process; however, our
layered controls and signing reduce this risk to an acceptable level.
```

```
**Related Controls:** Maps to NIST 800-53 SC-39 Process Isolation and
SC-7 Boundary Protection.
```

This kind of narrative cannot be autogenerated; it's the architects capturing knowledge for posterity and auditors.

2. **Threat Models and Risk Assessments** – While you might generate a skeleton (e.g., list of threats), the analysis and prioritization is manual. Write a structured threat model (like a STRIDE per component):

```
**Threat Model: API Gateway**

*Spoofing:* An attacker might impersonate the API client.
- **Mitigation:** OAuth 2.0 with JWTs, plus mutual TLS for internal
services.

*Tampering:* Malicious alteration of data in transit.
- **Mitigation:** TLS 1.3 for encryption, digital signatures on tokens/
payloads.

*Repudiation:* A user denies performing an action.
- **Mitigation:** All admin actions are auditable with signed logs (see
Audit Logging section).

*Information Disclosure:* Sensitive data exposure via the API.
- **Mitigation:** Field-level access controls, PII encryption at rest,
and data minimization in responses.

*Denial of Service:* Overloading the API to disrupt service.
- **Mitigation:** Rate limiting (60 requests/minute per client), auto-
scaling, and WAF rules for known attack patterns.

*Elevation of Privilege:* Exploiting the API to gain higher privileges.
- **Mitigation:** Role-based access control enforced on every request;
no direct admin actions without admin token.
```

Such analysis demonstrates to assessors that you systematically considered attacks. Keep the threat model updated whenever architecture changes (perhaps review it quarterly).

3. **Risk Register** – Document the top risks in a table with their mitigation status. This is typically manual because it involves judgment:

Risk ID	Description	Likelihood	Impact	Risk Level	Mitigation Status
RSK-001	Plugin escapes sandbox via shared memory exploit	Low	High	**Medium**	Controls in place (OS-level isolation, continuous monitoring). Re-evaluate after Red Team test Q4.
RSK-002	Compromise of API secret keys	Medium	High	**High**	Mitigated: HSM used for key storage, rotation every 90 days.
RSK-003	Supply chain attack (malicious dependency)	Low	High	**Medium**	Mitigated: Use pinned hashes, Dependabot alerts, periodic audits (Safety tool).
RSK-004	Zero-day in underlying LLM model	Unknown	High	**High**	Accepted: Cannot control model internals; rely on vendor patching SLA and have fallback service.

This shows you have a handle on residual risks and their treatments. Update this whenever a new significant risk is identified or an old one is resolved.

4. **Compliance Narratives** – For each required document (e.g., an SSP section, contingency plan summary, etc.), craft a narrative that references the automated evidence. For instance, in a section describing Access Control (AC family in NIST 800-53), you might write:

    > *Access Control Overview:* The system implements role-based access control (RBAC) with the principle of least privilege. There are four roles: User, PowerUser, Admin, and System. Permissions are defined in code (see :py:mod: `elspeth.core.auth`) and enforced on every API endpoint (see :doc: `API documentation <api/core>` for each endpoint's required role). Multi-factor authentication is required for all Admin actions. All access events are logged and monitored (AC-2, AC-6, AC-7 controls).

    And so forth, citing specific controls and how you meet them. These can't be fully autogenerated because they need to tell a story and convince auditors that you meet the intent of the control, not just the letter.

Tip: To ensure manual docs stay up-to-date, assign an owner or rotate responsibility for each major section. For example, one team member owns the threat model section and must update it if new threats emerge or architecture changes; another owns compliance mappings, etc. During each release cycle or sprint review, quickly sanity check if any change (epic, story) would require a doc update in these sections.

Linking Automated and Manual Sections

Ensure there is no wall between generated content and written content: - Reference code from manual docs (use Sphinx cross-references as shown). - In code docstrings or auto-generated sections, include references to manual discussions (you can even insert literal includes from files if needed to show a snippet of a policy or decision).

For example, you could include a snippet of the risk register CSV into the docs so it's visible, but maintain the single source CSV. Sphinx has the `csv-table` directive which could read it directly.

Another example: If using Bandit, you might maintain a custom config (`.bandit.yml`) that marks certain findings as not applicable. Document such decisions (with rationale) in the security docs. That ties tool configuration to documentation, again showing thoughtfulness.

---

4. SECURITY FRAMEWORK INTEGRATION

Accreditation involves mapping your implementation to prescribed controls and practices. Automation can help, but careful documentation is needed to make the case that you comply with each item.

NIST SP 800-53 and Other Frameworks Mapping

NIST 800-53 (Rev. 5) has families like Access Control (AC), Audit and Accountability (AU), Configuration Management (CM), etc., each with numerous controls (AC-1, AC-2, …). For a formal accreditation (FedRAMP Moderate, for example), you will need to provide an implementation statement for each applicable control and an assessment of how you meet it.

Strategy: Create a document (or section) that enumerates each control and maps it to evidence: - **Implementation Statement**: A short description of how the system addresses the control. - **Artifacts/ Evidence**: References to docs or files (could be within this docs site or external) that substantiate the implementation.

For example, for **AC-2 (Account Management)**: - *Implementation:* "Elspeth uses a centralized IAM service. User accounts are managed via an admin UI; creation requires approval of an Admin. Inactive accounts are automatically disabled after 90 days. See Section 5 of the System Security Plan for processes." - *Evidence:* "User account creation log (audit trail)【audit_system†L50-L60】, Admin guide on account management (DOC-XYZ), Code reference: :py:func:`elspeth.core.auth.create_user` which includes role assignment logic."

You might format this as a table in RST or as a definition list for each control.

FedRAMP Moderate will largely align with NIST 800-53, but also expects specific documents: - **System Security Plan (SSP)**: Often a huge document – but you can generate parts of it from your docs. The docs we are writing can serve as the basis for many sections of the SSP (the system description, the control implementations, etc.). Consider using a tool or template to automatically fill in the SSP document from the structured data you have in Sphinx (maybe via Jinja templates or a simple script). - **Policies and Procedures**: Some might be outside the scope of the code (like an Incident Response Plan). Ensure references in docs to external policies are provided (e.g., "See Incident Response Policy (POL-IR-1) in corporate policy repository for IR-4 requirement"). - **Continuous Monitoring (ConMon)**

documentation: Show how the tooling (Bandit, Safety, etc.) feeds into continuous monitoring. For example, mention that you have a monthly review of dependency scan results, etc.

CMMC (Cybersecurity Maturity Model Certification) if relevant, focuses on practices and processes at various levels. This might be more high-level but you can map the technical controls to those practices.

Automation Idea: Use tags or comments in code to assist mapping. For example, decorate functions or classes with a custom decorator or docstring tag like `# control: AC-6` (least privilege enforcement). Then have a script that scans the code for these tags to produce an inventory of where each control is implemented. This can populate your control mapping table. This way, if a control mapping changes (e.g., you decide a certain control is also addressed by a new module), updating the code annotation updates the mapping.

Automated Evidence Collection

When preparing for an audit, gathering evidence is time-consuming. You can automate some of this: - Archive CI pipelines results (test reports, SAST reports, etc.) for each release. These can serve as evidence that, for example, you have a static code analysis process (RA-5 control in NIST). - Use *screenshots as code*: e.g., write a script to hit the running system's API endpoints for status, config dumps, etc., and store the outputs in the docs or an evidence folder. For instance, evidence for "data is encrypted at rest" might include a snippet of the database configuration showing `storage_encryption: enabled`. - If using a GRC (Governance, Risk, Compliance) tool or dashboard, see if it has an API to fetch status or reports, and include that in documentation or at least link to it.

Example (semi-automated table):

```
.. table:: Sample Control Implementation Evidence

   ====================  ========================================
==============================
   **Control**           **Implementation Summary**               **Evidence
(links)**
   AC-2 Account Mgmt     Accounts created by Admins only;
- :doc:`User Management SOP <policies/user_mgmt>`
                         auto-disable inactive accounts           - Audit log
excerpt in :doc:`Audit Logs <security/controls>`
   SC-8 Transmission     All data in transit encrypted (TLS 1.3)  - Nginx
config snippet :ref:`nginx_tls_conf`
                         Internode RPCs over mTLS                 - Wireshark
capture (attachment)
   RA-5 Vulnerability    Automated dependency and code scans      - Bandit
report (Q4 2025): :download:`bandit_q4.html <_static/bandit-report.html>`
                         run on each commit; monthly review       - Safety
report (Nov 2025) :doc:`Dependency Status <security/controls>`
   ====================  ========================================
==============================
```

This table mixes manual text with live links to evidence. For instance, `bandit-report.html` was generated by CI and stored in `_static` as per our workflow, and we provide a download link for auditors to see the raw report.

Finally, consider **SA-15 (Development Process, Formal Methods)** from NIST 800-53 if you're aiming for a very high assurance level. If you did the formal spec work, you can check that box and provide the spec as evidence, showing you go above and beyond by using formal methods (which are actually recognized in some standards as a best practice for high-impact systems).

---

5. PLUGIN ARCHITECTURE DOCUMENTATION

The plugin system is a core part of this platform and likely the most scrutinized (since it executes untrusted or semi-trusted code). Documenting it well is essential.

Plugin Registration and Discovery

If you use an entry-point mechanism (like `importlib.metadata` entry points or the **stevedore** library used in OpenStack for plugin loading), document that process clearly:

- Where do plugins live (a directory, an entry point group `elspeth.plugins`)?
- How does the system find them on startup?
- What is the expected plugin packaging (e.g., each plugin is a Python package with a `setup.py` that declares an entry point)? Show an example entry point definition from `setup.py` or `pyproject.toml`:

  ```toml
  [project.entry-points."elspeth.plugins"]
  "security_scanner" = "elspeth_plugins.security:SecurityScannerPlugin"
  "data_validator" = "elspeth_plugins.validator:DataValidatorPlugin"
  ```

- Document the *registration security checks*: e.g., "On startup, the PluginManager iterates through all discovered plugins. It verifies each plugin's digital signature (each plugin package must be signed by our release manager key). Only if the signature is valid and the plugin is on the approved list, it is loaded into the runtime."

You can include a sequence diagram or flowchart (using Mermaid or text UML) showing the plugin loading process:

```
sequenceDiagram
    participant Core as Core System
    participant Plugin as Plugin Package
    Core->>Plugin: Discover via entry point
    Core->>Core: Verify plugin signature (PGP/SHA256)
    alt Signature invalid or plugin not allow-listed
        Core->>Core: Skip loading plugin (log event)
    else Signature valid
        Core->>Plugin: Load module in restricted namespace
```

```
        Plugin-->>Core: Register plugin capabilities
        Core->>Core: Store plugin reference and metadata
    end
```

(This can be included in `architecture.rst` or a dedicated `plugins.rst` in the API docs section.)

Plugin Interface Contracts

Document the interface that plugins must implement. If there is a base class or abstract class (like `PluginBase` or an interface definition using `abc.ABC` or `Protocol`), put that in the API docs and ensure it's clear what each method is for and any security-related expectations.

Example in code:

```python
class PluginBase(ABC):
    """Abstract base class for Elspeth plugins.

    Plugins extend this to add new functionality. **Security
considerations:** Plugins run with limited privileges. They should not
attempt to perform file or network I/O unless explicitly allowed by the
plugin API.
    """
    @abstractmethod
    def initialize(self, config: PluginConfig) -> None:
        """Initialize the plugin.

        Called once at plugin load. **Must not** have side effects outside
the plugin scope (no global state changes).
        """
        raise NotImplementedError

    @abstractmethod
    def execute(self, data: Any) -> Any:
        """Core method to process input data.

        This is run for each request that the plugin handles. It should be
stateless (no reliance on past calls).

        Returns:
            The result data or response.

        **Security:** All inputs are pre-validated by the core before calling
this. The plugin should still handle unexpected input gracefully. The output
will be post-validated by core.
        """
        raise NotImplementedError
```

Then in documentation, explicitly list the steps the core takes to isolate plugin execution: - e.g. "The core will catch any exceptions from `execute` and sanitize them." - "Plugins cannot directly call core internals; communication is via a provided `PluginContext` object that offers limited methods (logging, metrics)."

Security Boundaries & Enforcement

Have a section in `trust-boundaries.rst` or `architecture.rst` dedicated to plugin vs core boundary. This should cover: - Memory/namespace: e.g., "Plugins execute in the same process but within a restricted namespace: the built-in `__import__` is shadowed, dangerous builtins like `open`, `exec`, `eval` are removed or wrapped." - OS-level: e.g., "All plugins run under Linux cgroups with CPU and memory limits, and with a specific SELinux context that prevents certain syscalls." - Communication: e.g., "Plugins cannot initiate network calls directly; any network operation must go through core-provided APIs which enforce allowed domains and protocols." - Lifecycle: e.g., "If a plugin misbehaves (throws exceptions, takes too long), the core will terminate its execution and disable the plugin."

It's useful to present this as a bulleted list of controls:

```
**Plugin Isolation Controls Implemented:**

1. **Import Whitelist:** Plugin code can only import from a set of allowed
modules (math, datetime, etc.). This is enforced by overriding `__import__`.
2. **Read-Only Builtins:** Builtin functions that could modify interpreter
state or access I/O (like `open`, `__import__`, `exec`) are removed or
replaced with safe versions.
3. **Resource Limits:** Each plugin invocation runs with a time limit (e.g.,
2 seconds) and memory limit (e.g., 100 MB) enforced via a watchdog thread and
cgroup respectively.
4. **Capability Tokens:** If a plugin needs to perform an action like writing
a file or sending a request, it must call a core API with a token. The core
only honors requests with a valid token that matches a pre-approved action
for that plugin.
5. **Audit Logging:** All plugin actions (start/stop, requests made,
exceptions) are logged with plugin identifier and origin, making audit and
forensic analysis possible.
```

Link each of these to where they are implemented: - For example, "Import Whitelist" could link to the code file or function where you override `builtins.__import__`. - Or to an ADR if one was written for that decision.

Configuration and Policy

If the system has configuration for plugins (like enabling/disabling certain plugins or setting their permissions), document that as well. Possibly you have a `PluginConfig` model (maybe using pydantic as in the content provided). If so: - Auto-document the config schema (pydantic can output JSON schema; you could include that or describe fields in a table). - Provide an example config snippet in the docs. - Describe how config changes are applied (e.g., require restart, or dynamic reload).

Example Walkthrough

Consider adding a short end-to-end example in the docs: "**Example:** Installing and running a plugin" – this can tie together everything: 1. Developer writes a plugin implementing the interface. 2. They package it and sign it. 3. They drop it in the `plugins/` directory or install it so it's discoverable. 4. The system starts, discovers the plugin, verifies signature, loads it. 5. A user triggers the plugin via some action; the core calls `plugin.execute()`. 6. The plugin returns a result, core post-processes it, returns to user. 7. If plugin misbehaves, core isolates the failure (maybe skipping subsequent calls, raising an alert).

Walking through this narrative helps readers (and auditors) confirm that "ah, a malicious or malfunctioning plugin will be caught at step X and won't bring down the system."

---

6. ADVANCED DOCUMENTATION TECHNIQUES

Beyond the basics, there are advanced practices that can further strengthen your documentation (and the system itself). Use these as appropriate:

AI-Assisted Documentation (Use with Caution)

Tools like GitHub Copilot or GPT-based helpers can draft docstrings or even sections of documentation based on code. This can boost initial documentation speed but **always review for accuracy**, especially for security details. AI might make up rational-sounding but incorrect explanations.

Our Recommendation: Use AI suggestions to flesh out routine parts of docs (like summarizing what a function does), but do not rely on it for threat models or compliance statements. Those require precise, correct information. Always have a security engineer review and edit any AI-generated content. For instance, Copilot might help by suggesting a docstring for a function, but it won't know the security context unless clearly coded in function name or prior comments.

Formal Methods in Practice

As mentioned, formal methods can be integrated now rather than later: - If using **TLA+**: maintain a `.tla` specification in `docs/architecture/formal/`. Perhaps even include some high-level snippets of it in the docs for illustration. For example, if you specify the access control state machine, you could include a snippet:

```
---- MODULE access_control ----
VARIABLES users, roles, permissions, sessions

(* A user can only perform actions if a session with appropriate role exists
*)
ActionEnabled(u, action) ==
  \E r \in roles, s \in sessions: s.user = u /\ action \in permissions[r]
...
```

and then mention in English what this ensures. - If using **Alloy**: similar, include the model or critical invariants (Alloy finds counterexamples, which you can then mention if none found for a property). - The fact that you considered formal verification can be a competitive differentiator in an audit. It shows a maturity of process (in some frameworks like Common Criteria or certain ISO standards, formal methods are highly valued).

(Real-world note: Amazon Web Services routinely uses TLA+ for design verification of critical systems, reporting that it has found bugs that all other testing missed [1] *. While your system might not be as complex as AWS, even modeling one piece can increase confidence. If not now, at least the groundwork is laid.)*

Design by Contract and Runtime Enforcement

Consider using a *design by contract* library like `icontract` or Python's built-in `assert` statements to make the code self-documenting. For example, if a function assumes a certain input shape, put an assertion and also note that in docs. This way, if something ever violates it, it's caught during testing. Documentation-wise, Sphinx can pick up function annotations and docstring notes about these conditions.

For example:

```python
from icontract import require, ensure

@require(lambda user: user.is_active, "User must be active")
@ensure(lambda result: result is not None, "Result should not be None")
def get_user_data(user: User) -> dict:
    """Retrieve data for an active user.

    Preconditions:
      - User must be active (else no data is returned).
    Postconditions:
      - Result is never None (if user is active).
    """
    ...
```

This appears in documentation and also enforces at runtime during testing. It's an advanced technique that aligns code and docs closely.

Property-Based Testing as Documentation

Security properties can be tested using frameworks like Hypothesis (for Python). When you write a property-based test, include a docstring that reads like a specification. For instance, for password hashing you might write:

```python
@given(st.text(min_size=1))
def test_password_hashing(password):
    """Property: Verifying a correct password should succeed, and a wrong
password should fail."""
    assume(len(password) < 100)  # some limit
    hashed = auth.hash_password(password)
    assert auth.verify_password(password, hashed)
    # Try a slightly modified password
    assert not auth.verify_password(password + "x", hashed)
```

This test doubles as documentation of the expected behavior of the auth system. You could even include such examples in the docs (maybe not the Hypothesis code, but the property description: "We expect that for any password, if you hash it and then verify the same password, it succeeds, but any different password fails. This is tested automatically.")

Architecture Decision Records (ADRs)

We have included an ADR folder. Using ADRs is a great way to document the evolution of the system's architecture and security decisions. They are essentially mini-design docs focusing on one decision.

In the context of security, ADRs might cover: - Choice of tech stack (e.g., "Using FastAPI vs Flask vs Django for the API"). - Authentication method (e.g., "Decided on OAuth2 with JWTs"). - Data encryption approach (client-side vs server-side). - Threat model scope decisions (e.g., "We assume the cloud provider is trusted – an assumption logged on date X").

Each ADR should have: - **Context**: What problem or question was being addressed. - **Decision**: What was chosen and why. - **Status**: Accepted/Rejected/Superseded. - **Consequences**: Pros/cons of the decision, especially on security. - **References**: Any research or standards considered.

Since we already plan to include ADRs in the docs (via `adr/index` in toctree), make sure to link relevant ADRs from the main text: - In the plugin section, link to "ADR-007 Plugin Isolation via Namespace Restrictions". - In the architecture, link to "ADR-001 Tech Stack" if it includes security rationale for using Python etc. - In threat model or controls, if an ADR covers a compensating control, reference it.

This shows a trace from high-level decisions to implementation, which is excellent for audits (it demonstrates a rational process, not just ad-hoc decisions).

---

7. TOOLS FOR SECURITY DOCUMENTATION AUTOMATION

Rounding out the toolkit, here are some additional tools and techniques that can assist in creating or managing specific parts of the security documentation:

- **Threat Modeling Tools**: Consider using tools like OWASP Threat Dragon (open source) or Microsoft Threat Modeling Tool to create threat model diagrams (DFDs with threat annotations). They often allow exporting to images or embedding in documentation. However, maintain the textual threat analysis in the docs for searchability and linking. The diagrams can be supportive visuals.
- **Data Flow Diagramming**: For data flow diagrams (if Mermaid is not sufficient or you want more formal notation), consider using draw.io or Visio to create DFDs that include trust boundaries. You can export these as SVG/PNG and include in docs. Ensure each DFD is explained with text.
- **Attack Surface Tools**: Tools like Nmap or OWASP ZAP can be used to scan your running application (especially if it's a web service) to enumerate open ports or basic vulnerabilities. The results can be used as evidence that you tested your surface. For instance, include a snippet from an Nmap scan showing only the intended ports are open.

- **CodeQL**: GitHub's CodeQL can be used for customized queries. If you have enterprise GitHub, you might incorporate CodeQL security queries and then reference the results (or the fact that "CodeQL analysis passes with zero critical findings").
- **Documentation Coverage**: A fun meta-metric: ensure that every module or significant class has *some* documentation. You could write a small script to verify that each module is mentioned in the Sphinx toctree or in an autodoc. This can prevent parts of code from being completely forgotten in docs.

Remember that adding more tools means more maintenance. Only add what your team can commit to keeping up. The core described earlier is already a lot; these are optional enhancements.

---

8. CASE STUDIES: EXEMPLARY PROJECTS

Looking at how other projects handle documentation and security can provide valuable guidance:

- **Django (Web Framework)** – *Security documentation excellence*: Django's docs include a dedicated "Security" topic guide covering everything from common vulnerabilities to how Django's features address them. They use Sphinx [5]. Takeaway: Keep a section of your docs specifically for "security considerations" of using or deploying the system, as Django does. Also note how Django discloses security issues (their responsible disclosure policy) – if your platform will be deployed externally, consider stating how vulnerabilities in it would be handled.

- **OpenStack** – *Extensive use of automated docs*: OpenStack has a separate security guide (built with Sphinx) and each project has thorough docs. They use **stevedore** for plugins and have documentation on how that works. Takeaway: The OpenStack Security Guide (though for a much bigger system) shows how to structure a security document that includes conceptual info and best practices [3]. Also, OpenStack's use of a plugin system (stevedore) can inform how you explain your plugin architecture.

- **Bandit (PyCQA)** – *Tool documentation*: Bandit's own documentation (on Read the Docs) is an example of documenting a security tool. It lists each test plugin (with CWE references), how to configure it, etc. For our purposes, note how they present security findings. Perhaps mimic that clarity when documenting what types of issues your system's security controls address.

- **FastAPI** – *Modern docs and interactive elements*: Although FastAPI's focus is not security, its documentation (using MkDocs Material) is very user-friendly, with lots of examples and even interactive docs. Consider a small section in your docs like "Security API Usage Examples" where you show, for instance, how an admin would rotate a key using your system's API, or how an auditor might retrieve an audit log. Real-world usage scenarios can often clarify how all the pieces (auth, audit, plugins) come together.

- **OWASP Projects** (ASVS, Cheat Sheets) – These are not code projects but documentation projects that are well-structured. E.g., OWASP ASVS is essentially a checklist of controls with descriptions. If relevant, you can align some of your content with ASVS categories (like "Authentication", "Access Control", etc.). If an auditor sees familiar structure (like ASVS chapters), it might resonate.

Each of these cases provides inspiration for structure, tone, or thoroughness. We can adapt ideas, but ensure the documentation remains tailored to *our* system, Elspeth.

---

9. REFERENCE ARCHITECTURE & DOCUMENTATION REPOSITORY

Bringing it all together, here's an example of how the documentation repository could look when complete, and how it is maintained over time:

```
docs/
├── user-guide/
│   ├── index.rst                # User-facing documentation index
│   └── ...                      # Other user docs
├── architecture/
│   ├── conf.py                  # Sphinx configuration for architecture docs
│   ├── Makefile                 # Make commands to build docs
│   ├── index.rst                # Security/architecture docs index
│   ├── security/
│   │   ├── architecture.rst     # System architecture overview (manual +
diagrams)
│   │   ├── trust-boundaries.rst # Explanation of trust zones
│   │   ├── data-flows.rst       # Data flow diagrams and explanations
│   │   ├── threat-model.rst     # STRIDE analysis of major components
│   │   ├── controls.rst         # Catalog of security controls implemented
│   │   ├── risk-assessment.rst  # Risks, mitigations, and risk matrix
│   │   └── compliance/
│   │       ├── nist-800-53.rst  # Detailed control-by-control implementation
│   │       ├── fedramp-moderate.rst # Specific notes for FedRAMP
│   │       └── cmmc-level2.rst  # Specific notes for CMMC if needed
│   ├── api/
│   │   ├── core.rst             # Autodoc of core package
│   │   ├── plugins.rst          # Autodoc of plugins package
│   │   ├── datasources.rst      # Autodoc of datasources package
│   │   └── tools.rst            # Autodoc of tools/utilities
│   ├── adr/
│   │   ├── index.rst            # List of ADRs
│   │   ├── 001-tech-stack.md
│   │   ├── 002-auth-method.md
│   │   ├── 003-data-encryption.md
│   │   ├── 007-plugin-isolation.md
│   │   └── ... etc.
│   ├── diagrams/
│   │   ├── generated/           # Auto-generated images (not all checked in)
│   │   └── manual/              # Any manually created diagrams
│   └── _static/
│       ├── bandit-report.html   # Security scan outputs for reference
│       ├── semgrep-report.json
│       └── coverage.txt         # Maybe test coverage summary
└── ...
```

Note: The `user-guide` and `architecture` folders could either be separate Sphinx projects or a single one. If single, the `conf.py` would be at `docs/conf.py` and the toctree would include both user and architecture docs. If separate (as shown), you might have two different documentation outputs. Since the security docs might be internal or not for all users, separating can be wise.

Maintenance Schedule

To keep this living documentation accurate and effective, establish a regular review cadence:

- **Daily/Continuous:** As part of the CI pipeline, if a build fails due to doc issues (e.g., broken links, autodoc errors) or a security scan fails, treat it as urgent just like a failing unit test. This ensures small problems are fixed immediately.
- **Weekly:** Have a quick team sync or checklist: "Did anyone merge a PR that affects docs content?" If yes, ensure those sections are updated. Also review new Bandit/Safety findings weekly; if the tool updated rules and flags new issues, address code or mark them with a justification in docs (this shows you actively manage security findings).
- **Bi-Weekly/Sprint Review:** Reserve time to update threat models or risks if new features were added. E.g., a new plugin type was introduced – does it introduce new threats? Add them to threat-model.rst.
- **Monthly:** Run a full **link check** (`make linkcheck` in Sphinx) and fix any broken references or outdated external links. Also, if you have diagrams or models, ensure they still reflect reality (the automated ones should; any manually drawn ones need review).
- **Quarterly:** Do a deep maintenance pass:
- Update dependencies of doc tools (Sphinx, etc.) and ensure the docs still build fine.
- Revisit compliance mappings in light of any regulation changes or internal policy updates.
- Perform an **internal audit**: pick a few controls from NIST 800-53 and actually verify that the documentation and evidence you have would satisfy an auditor. Adjust as needed.
- If formal specs exist, maybe re-run the model checker on the latest design or see if code changes necessitate spec changes.
- **Annually:** Major review:
- Rewrite sections that have gotten bloated or unclear.
- Consider feedback from any real audits or pen-tests conducted.
- Update the overall security plan to align with corporate or product changes.
- (If aiming for higher certifications) consider if more formal processes or documents should be added.

By treating documentation as a continuous part of development (and not a one-time task), it remains accurate and a real asset rather than a liability.

Also encourage all team members to view and use the documentation. For example, new hires should be onboarded via reading these docs. If they find inaccuracies or have questions, that's a trigger to improve the docs. A culture of documentation pays off when the pressure is on during an accreditation review.

---

10. RESOURCE LIST

Official Documentation & Tools: - Sphinx Documentation – Comprehensive guide to Sphinx usage and extension development. - Read the Docs – Although primarily for hosting, their guides cover structuring docs, automation, etc. - Bandit Documentation – List of built-in tests and how to configure output. - Semgrep Rules Registry – Pre-written security rules you can leverage. - Pyreverse (Pylint) Docs – Info on pyreverse usage (part of Pylint). - pydeps on GitHub – Usage examples for generating graphs. - PyCG on

GitHub – Instructions for generating call graphs and interpreting output. - icontract – Design-by-contract library for Python. - Hypothesis – Property-based testing library.

Security Standards & References: - NIST SP 800-53 Rev.5 Controls – The master list of security controls (useful for ensuring you covered everything). - FedRAMP Moderate Baseline – Specific requirements on top of NIST for cloud systems. - OWASP ASVS 4.0 – Good checklist of application security requirements (map some of these to your controls to ensure coverage). - CWE Top 25 (2024) – Keep an eye on common weaknesses to ensure your tooling and documentation address these where relevant.

Learning and Community: - *Write The Docs* community – tips on maintaining documentation within development workflows. - *Security Stack Exchange* and OWASP Slack – great for asking specific questions when documenting security aspects (like "what's the best way to document threat models?"). - *Microsoft Azure Security Documentation* – as an example of well-structured cloud security docs (useful if looking for another perspective on organizing such content).

---

11. COST-BENEFIT ANALYSIS

Investing in documentation and tooling has upfront costs, but pays dividends by preventing costly delays in accreditation and reducing security incidents.

Initial Costs: - Developer time to set up Sphinx, CI, and write initial docs: ~12 weeks of one developer (or spread across team) – if we value that at say $120k/yr, that's ~$30k in labor. - Training and ramp-up: say $5k worth of time for workshops and self-learning. - Tooling: All recommended tools are open-source. If you opt for any commercial add-ons (perhaps a paid Semgrep Team plan or a fancy diagramming tool license), that could be a few thousand dollars, but let's assume open-source for now. - Total initial: *approximately $35k in effort* (mainly labor).

Ongoing Costs: - Maintenance of docs (4-6 hours/week across team): ~0.15 of an FTE, ~$15k/year. - Continuous integration resources: minimal monetary cost (GitHub Actions minutes, etc., which are usually low for documentation). - Possibly subscriptions if you later opt for something like an automated compliance tool (could be $10-20k/year, but optional). - Total annual ongoing: *~$20-25k/year* in effort and minor expenses.

Tangible Benefits: - **Faster Accreditation:** Well-prepared documentation can cut down the back-and-forth with assessors. If accreditation normally takes 3 months of intense effort, you might cut it to 1 month. That could save perhaps $20k in staff time immediately and also allow the product to go to market faster (which could be worth even more in business terms). - **Avoided Re-engineering:** By discovering design issues early (thanks to thorough analysis and possibly formal methods), you avoid costly rework. A security flaw found during an audit might delay launch by months. If our documentation and analysis prevent even one such major redesign, that could be saving $50k+ and significant reputation risk. - **Reduced Incidents:** A documented secure design, with continuous checks, reduces the chance of a breach. Avoiding one medium security incident (which could easily cost $100k in incident response, not to mention reputation damage) justifies a lot. Our robust process should reduce incident likelihood (though cannot eliminate it). - **Developer Efficiency:** New developers or contributors ramp up faster with good docs, saving onboarding time (which is money). Instead of pinging senior engineers with basic questions, they can read the docs. If it saves each of, say, 5 developers two weeks of ramp-up time, that's another ~$25k saved. - **Audit Confidence:** In formal audits, organizations that present clear documentation often face fewer scrutiny hours (auditors don't need to pull as hard to get info). This can reduce external auditor cost if you're paying for their time, and certainly reduces internal stress.

All told, within the first year, the benefits (in saved time, avoided issues) could easily exceed $150-200k in value, far outweighing the ~$35k initial and ~$20k ongoing costs.

Additionally, there's an **intangible but real** benefit: **trust**. Having this level of documentation and process can impress customers or partners (especially for a security product). It shows maturity. That can be the difference in winning a contract or not in some cases.

In summary, the ROI is significant – on the order of **5-10x return** on the investment within the first year alone, and compounding thereafter as the system scales and more audits occur.

---

12. FINAL RECOMMENDATIONS

Prioritized Action Plan

Immediately (Week 1): - **Set up the documentation repository** as outlined. Initialize Sphinx in the `docs/architecture` folder, get a basic index and conf.py working. (Have a "Hello World" doc build in CI by end of week 1.) - **Draft the outline** of all sections in the security docs (even if empty). This creates placeholders so others can contribute in parallel. - **Install and configure core tools**: Bandit, Semgrep, pyreverse, etc. Test run them on the codebase to ensure they work and to get initial reports. - **Plan formal method use:** Identify one component (maybe the access control) and see if someone (or a consultant or a formal methods savvy team member) can start a simple TLA+ model. Order any necessary books or allocate training time if team is new to this.

Short Term (Months 1-2): - **Implement CI/CD for docs and scanning**. Make sure every commit triggers documentation build and security tests. This might involve some trial and error (e.g., tuning Bandit rules, figuring out how to publish artifacts). - **Begin filling in documentation content:** Start with the high-level architecture and security controls, since those are fresh in mind and critical for guiding everything else. Also, populate the compliance matrices early, as that will reveal if you have gaps (e.g., "Oh, we have no control listed for CA-7 continuous monitoring – maybe our process needs something there"). - **Conduct a mini-internal review at end of Month 2:** Pretend to be an auditor and see if you can navigate the docs to answer basic questions like "How is data encrypted?" or "How are admin actions logged and monitored?". Any question that's hard to answer indicates a documentation gap.

Mid Term (Months 3-6): - **Complete threat modeling workshops** with the team for each major component and document the outcomes. Doing this collaboratively not only produces good docs but also educates the whole team on the security posture. - **Finalize the formal specification (if doing)** by month 6. By this point, you should know if formal methods are yielding value. If yes, include the results in docs; if it's not proving useful, document why and move on (the attempt still isn't wasted – it likely improved understanding). - **Prepare for a pre-audit**: Have an outsider (or someone from another team) do a "mock audit". They get the documentation as if they were an assessor and they ask questions or find holes. Use this to refine unclear sections. - **Keep polishing automated parts**: maybe by now you'll refine how diagrams look, or adjust which sections are auto vs manual as you see what works best.

Long Term (6-12 months): - By month 6, ideally, you're in a state to request formal accreditation (if that’s the goal). From 6-12, it's responding to auditor questions, providing additional evidence, and possibly implementing any improvements they suggest. - **Continuous improvement**: Establish the habit that whenever a security-related change is made in code, a corresponding doc update is made if needed. Make this a checklist item in code reviews. - **Knowledge transfer**: If one person was driving this, ensure the whole team is familiar with the setup. Share the load of writing ADRs or updating sections. A bus factor of 1 on documentation is a risk.

Success Metrics

How do we know this effort is succeeding? Track some metrics: - **Documentation Coverage**: Percentage of modules with docs, or ratio of docs pages to code (not exact science, but you can track pages or word count over time ensuring it's growing meaningfully). - **CI/CD Health**: Docs build success rate. Aim for near 100% (if failing, it's usually broken links or warnings; those should be fixed promptly). Also track that security scans in CI remain green (or issues are promptly triaged in the backlog). - **Audit Findings**: If after an audit you have only minor documentation-related findings (or none), that's a huge success indicator. Strive for 0 "documentation missing or insufficient" findings. - **Team Engagement**: Are developers actively contributing to docs? This can be measured by docs git commits by different team members. Aim to have everyone contribute, not just one writer. - **Time to Onboard New Dev**: If before it took 2 weeks for a new hire to feel comfortable, and now with docs it takes 1 week, that’s a quantifiable improvement. Get feedback from new folks specifically on the helpfulness of docs. - **Security Posture**: Fewer security issues reported in testing or by users. This one's hard to tie directly to docs, but good docs correlate with well-thought design which correlates with fewer issues. If you do bug bounty or penetration tests, track the number of issues or their severity trending down over releases.

Key Success Factors

1. **Management Buy-In** – It's important that leadership understands the value of this effort. If PMs or execs see documentation as "overhead," remind them of the accreditation dependency and ROI. If they’re on board, you'll get the time and resources needed.
2. **Team Culture** – Foster pride in documentation. Celebrate a well-written ADR as you would a tough piece of code. Make security everyone's responsibility, and by extension, its documentation as well.
3. **Start Simple, Iterate** – The plan is comprehensive, but it can start small. It's okay if initially some sections are stubs or some tools aren't fully utilized. Incremental progress (with each sprint leaving docs a bit richer) is better than trying to do it all perfectly then never updating it.
4. **Automation Balance** – Use automation to handle the grunt work, but always review automated outputs. For example, if diagrams are too cluttered, intervene by splitting into multiple diagrams or adding manual annotations. If Bandit flags something minor repeatedly, either adjust it or document why it's acceptable – don't let the automated tools create noise that people start ignoring them.
5. **Feedback Loop** – Encourage feedback from everyone who reads the docs: developers, auditors, even friendly customers if any get access to certain parts. If something is confusing or missing, that’s valuable input to improve it.

With these factors in place, the documentation and security program will not stagnate. It will evolve with the system, and continue to be a valuable asset rather than a snapshot that decays.

CONCLUSION

This comprehensive strategy provides a proven path to achieve a high-quality, accreditation-ready documentation set for the Elspeth platform, without sacrificing developer productivity. By using Sphinx as the backbone and integrating visualization and analysis tools, we ensure the documentation is both human-friendly and technically rigorous. The inclusion of formal methods for critical components, planned from the beginning, adds an extra layer of confidence in the system's design integrity.

Adopting this approach requires an initial investment of time (roughly 3 months of concerted effort), but the payoff is substantial: smoother accreditation audits, easier maintenance of security knowledge, and enhanced trust with stakeholders. In essence, we treat **documentation as code** – living, reviewed, and versioned – and **security as a first-class feature** of the platform.

Next Steps: Kick off the Week 1 tasks: set up the docs repo, schedule a team docs kickoff meeting, and start populating the skeleton. Simultaneously, ensure CI is ready to catch doc and security issues. By laying this groundwork now, we prevent a scramble later and instead approach the formal accreditation with confidence, backed by automated evidence and clear, structured documentation.

---

[1] How Amazon Web Services Uses Formal Methods – Communications of the ACM
https://cacm.acm.org/research/how-amazon-web-services-uses-formal-methods/

[2] [5] Writing documentation | Django documentation | Django
https://docs.djangoproject.com/en/dev/internals/contributing/writing-documentation/

[3] RST conventions — Documentation Contributor Guide documentation
https://docs.openstack.org/doc-contrib-guide/rst-conv.html

[4] CWE Coverage for Python
https://www.byteplus.com/en/topic/499981