embarc
heise conferences

architektur SPICKER

Essential knowledge for software developers squeezed into a few pages

FIND MORE CHEAT SHEETS ONLINE:
Free pdf-Download of the architecture cheat sheet collection:
www.architektur-spicker.de

NO. 15

Documentation-as-Code in Practice

Using the same tools and processes as in software development enables efficient documentation maintenance. This cheat sheet explains the concept in detail and helps you document your initiatives.

IN THIS ISSUE

What's it all about? (challenges)

- Existing documentation is cumbersome and outdated. How can you create and keep it current with your familiar tools?
- A picture is worth a thousand words. Which simple approaches and tools help you create effective diagrams?
- Docs-as-Code offers many tools, formats and best practices. How do you choose the right setup for your context?
- Over time, implementation and documentation tend to drift apart. How does the development team keep them aligned?

What is Docs-as-Code?

"Treat Documentation as Code": This approach treats documentation just like source code.

Typical characteristics

- Document in plain-text format
- Stored in version control
- Seamless integration with development tools
- Automated, regular documentation builds
- Content generation to reduce redundancy
- Team collaboration is improved
- Continuous improvement

Maturity model for Docs-as-Code

Progressive adoption of proven software engineering practices improves documentation quality and efficiency in your projects, enhances team collaboration and enables a seamless and transparent documentation culture.

5 Align with implementation

Make architecture concepts explicit and visible in the source code, derive and record **architecture rules** from them, and regularly verify them automatically as tests from the documentation.

This turns documentation into a living artifact!

4 Continuously improve documentation

Address the different target groups and their requirements, and regularly incorporate **feedback**. Integrate documentation creation and maintenance into **team processes**.

- peer reviews
- pull requests
- pair/ensemble documenting

3 Use and integrate tools

Avoid unnecessary context switches and create documentation with typical **development tools**. Use **automation** and **regularly build results** in CI/CD pipelines.

Use or integrate with:
- development environments
- build management tools
- command-line tools

2 Handle graphics efficiently

Avoid overusing UML tools and commercial graphics software, instead **lower the entry barrier** by using lean, free tools.

Go one step further with text-based formats (Diagrams-as-Code).

1 Lay the foundation for Docs-as-Code

Start with **markup languages**, organize documentation into modules, and create results tailored to specific audiences. Store all content in **version control systems**.

- avoid WYSIWYG approaches
- avoid redundancies
- do not bury content in a wiki or file-sharing platform

0 Document conventionally

Standard structures such as **arc42, C4 and ADRs** are used, but using conventional tools can demotivate teams and even **slow down documentation** processes.

The foundation for Docs-as-Code

With just a few core concepts, formats, and tools, you can get started with Docs-as-Code with minimal effort.

Markdown vs. AsciiDoc:
https://www.embarc.de/en/markdown-vs-asciidoc/

Lightweight text formats

1 **Use markup languages (e.g., Markdown, AsciiDoc, ...)** to write documentation without distractions, just like composing an email (as-code).

Filesystem-based storage

2 **Store your content directly in the filesystem, without any detours (no wiki, no WYSIWYG tool).** This allows you to search it at any time using standard operating system tools and edit it directly in your IDE or a text editor.

Version control

3 **Put your documentation under version control (e.g. Git), just like source code.** This allows you to easily version and release it in sync with your software artifacts, as well as track changes in the history ("diff").

Offline capability

4 **Using Git, you can also edit your documents offline and synchronize them with your team later.** This means you're not dependent on access to a central system (e.g., a wiki).

Part of the build process

5 **Render the content in different output formats, such as HTML, PDF, or a wiki** export, making it accessible to all stakeholders, including non-technical ones.

6 **Build the documentation continuously and automatically.** This way, you always have the latest version at hand and can easily integrate documentation into your review processes.

Continuous generation

7 **Generate parts from models (from source code, database schemas, API descriptions, ...).** This avoids manual copying and tedious synchronization (single source of truth).

Content generation

8 **Modularize your documentation** to edit it in smaller, redundancy-free units and recombine it later as needed.

Modular documentation

Flexible output

9 **Render content-specific documents** to deliver precise, targeted results to different stakeholder groups.

From Traditional Graphic Formats to Diagrams-as-Code

Diagrams can represent complex concepts, processes and structures far more compactly and comprehensibly. Tools should support us as effectively as possible in creating easily maintainable visuals for our documentation.

Good to know: **draw.io stores vector information within the bitmap metadata (JPG, PNG). The files can be viewed and embedded directly, while still remaining editable in their original form.**

	Conventional diagram creation	Lightweight, free tools	Diagrams-as-Code
Typical characteristics	• often commercial • extensive, but sometimes confusing feature set • images always have to be exported	• open source or free to use • modern, user-friendly UI • platform-neutral, often web-based • widely used in the community • partly collaborative features	• text formats for describing graphics • automatic diagram layout • partial support for UML, ... • initial DSL approaches for model maintenance, including validation
Common examples	• Microsoft Visio, Adobe Illustrator, PowerPoint (**vector graphics**) • UML tools such as Magic Draw, Enterprise Architect **(model-based)**	• draw.io/diagrams.net • other alternatives: yEd, Gliffy, Lucidchart • digital whiteboards: Miro, Mural, Conceptboard	• PlantUML, Mermaid • Vega/Vega-Lite, Ditaa • Structurizr • Kroki
Strengths and challenges	**Vektorgrafiken** ⊕ can be adjusted at any time ⊕ infinitely scalable size ⊕ SVG as a common exchange format ⊖ generic, not specifically designed for architecture documentation ⊖ additional effort to export into standard formats (*.jpg, *.png, ...) **Model-based tools** ⊕ consistent views and redundancy-free model ⊕ generation of source code, API descriptions, DB schemas, ... ⊖ high learning effort ⊖ often expensive, with shared license usage	⊕ feature set usually sufficient for typical architecture diagrams ⊕ web-based solutions can be embedded in wikis (Confluence) and IDEs (IntelliJ, Visual Studio Code) ⊕ a large number of available symbol palettes ⊖ confusing market ⊖ vendor lock-in poses a risk	⊕ easy to write and human-readable ⊕ quick to create and easy to update ⊕ editable with text editors or IDEs ⊕ can be automatically generated from data ⊕ easy to version, compare, and merge ⊖ layout issues with large diagrams ⊖ integration with model-based approaches is still not widely used or mature

As a general rule: vendor-specific file formats can lead to vendor lock-in and additional effort. Using widely accepted exchange formats (such as SVG) or text-based solutions avoids the problem.

When text-based diagrams don't deliver satisfactory results or are unnecessarily complicated to create

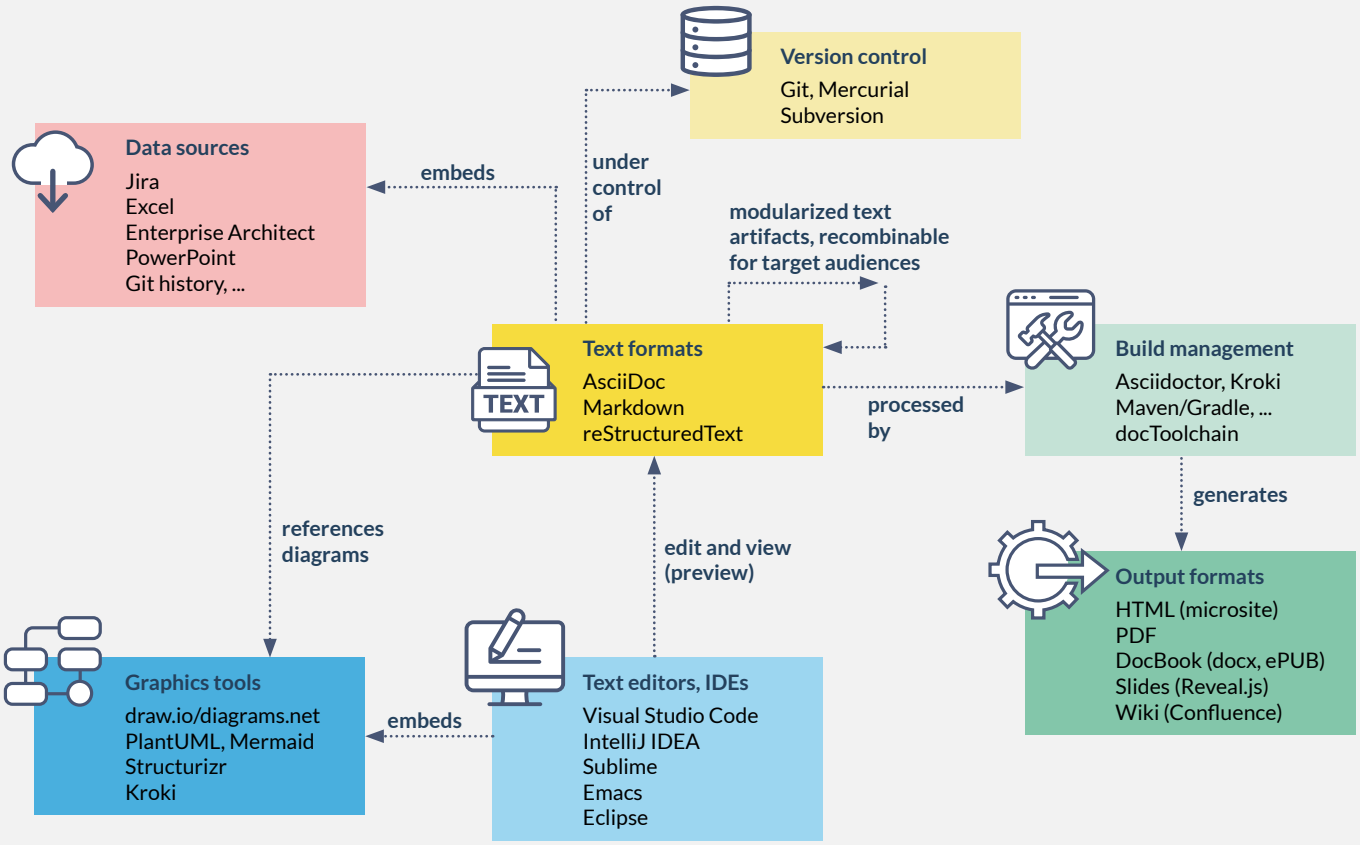
1 Simplicity and efficiency of Diagrams-as-code, along with good integration into docs-as-code tools and methods

2

3 For specific requirements, particularly model-based approaches

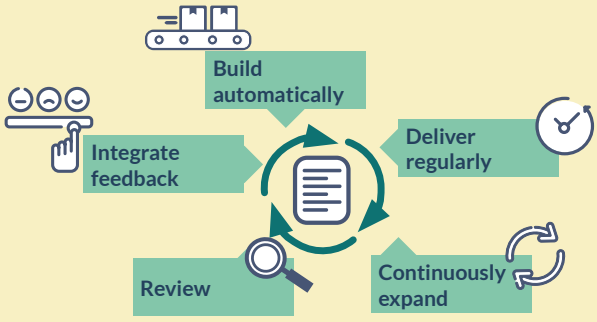
Implementing and efficiently integrating tools

This conceptual diagram illustrates the interrelationships between the concepts, artifacts, and tools of Docs-as-Code.



Create documentation continuously

To ensure high adoption among all stakeholders, you should integrate the build and maintenance of documentation into your continuous development processes.



Activity	Benefit	How to?
Build automatically	No manual effort, fewer errors, consistent quality	Embed in existing build management processes, process in the CI/CD pipeline
Deliver regularly	Centralized location with the latest version, easy to reference, eliminates distribution of documentation artifacts via email	Tag documentation with corresponding version numbers; store in an accessible location for all stakeholders, e.g. as an HTML microsite or PDF
Continuously add to	Boy Scout Rule: continuously clean up documentation and add new aspects	Documentation work becomes part of the Definition of Done; technical user stories include the update
Review	Share knowledge, detect errors early to achieve consistent high quality	Review documentation changes during code reviews (pull requests) and request missing content
Integrate feedback	Considering stakeholder requirements and reader preferences enhances acceptance	Evaluate improvement suggestions, incorporate them regularly and purposefully, and communicate updates to project participants

Pitfalls in practice

**Even with Docs-as-Code, not everything is perfect.
We'll show you how to deal with the challenges.**

Context	Problem	Suggested solutions
Lack of technical background	How do we involve employees without a development-related background?	• have documentation written primarily by developers/architects • import data from familiar formats/tools (Excel, PowerPoint, Jira, Enterprise Architect, etc.) • provide timely availability and direct read access for all stakeholders • create simple feedback options and web-based WYSIWYG editing capabilities for non-technical project participants as well
Feedback cycle	Despite a heterogeneous toolchain, how do we integrate all participants (authors, reviewers, readers) into the workflows?	• establish documentation updates as an acceptance criterion in user stories • enforce regular reviews, e.g. as part of code reviews • automate documentation generation and delivery (via CI/CD pipeline) • synchronize delivery with the software artifacts • use standardized interfaces and formats
Establish D.R.Y. (Don't Repeat Yourself)	How do we avoid redundancies within the documentation and with other models?	• modularize documentation (small text files) based on standard structures • continuously generate content from code, APIs, database schemas, ... • embed information from existing sources (Excel, Jira, Git, Enterprise Architect, ...) • executable documentation (embed and regularly check architecture rules)

Executable and living documentation

Creating documentation is only one side of the coin. How do we ensure that the intended audience actually reads the documentation, provides feedback, and that this feedback is incorporated into ongoing revisions and updates?

- Create artifacts tailored to target groups
- Synchronous delivery enables always up-to-date availability
- Embed code snippets and API calls
- Integrate architecture rules and fitness functions into the documentation
- Enable reader feedback

Further information

Selected references + tools

- Hitchhiker's Guide to Documentation as Code (https://docs-as-co.de/)
- docToolchain - an open-source tool that simplifies the creation and maintenance of technical documentation (https://doctoolchain.org/)
- kroki.io creates diagrams from textual descriptions (https://kroki.io/)

Blog posts

- What is Docs-as-Code? (embarc.de/en/what-is-docs-as-code/)
- Quick start with docToolchain (embarc.de/en/doctoolchain/)
- Markdown vs. AsciiDoc (embarc.de/en/markdown-vs-asciidoc/)

The author of this cheat sheet

Falk Sippach | Contact: falk.sippach@embarc.de

We look forward to your feedback: spicker@embarc.de

https://www.embarc.de/en/cheat-sheets/

embarc
https://www.embarc.de
info@embarc.de

heise conferences
https://heise-conferences.de
konferenzen@heise.de