Comprehensive Research Report: Verification and Expansion of the Self-Healing Spring Boot Microservice POC with GenAI-Driven Root Cause Analysis

---

Introduction

In the evolving landscape of cloud-native application development, the demand for resilient, autonomous, and self-healing systems has grown dramatically. Modern microservice architectures-exemplified by Spring Boot-need to not only detect anomalies and faults in real time but also to autonomously analyze root causes, document findings, and, where feasible, remediate problems with minimal human intervention. By integrating lightweight Generative AI (GenAI) into local service stacks, it is now possible to bridge the gap between traditional system observability and semi-automated, AI-driven repair. This research report provides a thorough, end-to-end validation and enhancement blueprint for a self-healing system proof-of-concept (POC) consisting of a Spring Boot microservice that:

- Handles HTTP POST requests with validation and persists data in local JSON files.
- Integrates a local, lightweight GenAI model to understand logs and traces, detect issues, generate human-readable root-cause and fix reports, store those locally, and apply changes after approval.
- Runs entirely on a Mac, requiring no external database.
- Includes architecture, implementation plan, detailed component analysis, recommended improvements, and a text-based architecture diagram aligned with contemporary conventions.

Leveraging insights from 'Self-Healing Service POC Architecture - Claude', primary literature, live code repositories, and authoritative industry resources, this report critically inspects every component, identifies gaps, addresses architectural and security considerations, examines appropriate design patterns, and expands upon the original intent to create a robust, extensible POC for self-healing microservices.

---

1. Overarching Principles of Self-Healing System Architecture

Self-healing systems are frameworks capable of detecting, analyzing, and autonomously repairing faults with minimal human intervention, ensuring high reliability, continuous availability, and rapid recovery from both expected and unforeseen failures. The primary pillars of such systems include comprehensive observability (logs, traces, metrics), intelligent diagnostics (root-cause identification), safe remediation or rollback, and robust approval workflows.

Key characteristics and benefits include:

- **Autonomous detection, diagnosis, and remediation**: Systems continuously monitor their own state, identify anomalous behavior, analyze root causes (often with AI), and execute predefined or dynamic remediation steps[1].
- **Reduced downtime and operational overhead**: Automation of routine error handling relieves operators and engineers, ensuring higher user satisfaction and lower maintenance costs.
- **Resilience and scalability**: System redundancy, failover patterns (circuit breaker, bulkhead), and dynamic configuration management mitigate the impact of individual faults[2].
- **Incorporation of AI/GenAI**: Enhances anomaly detection, root-cause analysis, and intelligent reporting, driving proactive, rather than reactive, system health strategies[4].
- **Human-in-the-loop assurance**: For sensitive or code-modifying actions, manual approval steps ensure safety, security, and trustworthiness[5].

---

2. POC Architecture Overview and Text-Based Flow

A robust, locally runnable self-healing POC-centered on a Spring Boot microservice, file-based JSON persistence, and a lightweight GenAI engine-should embody the following flow:

[Client]
|
HTTP POST
|
[Spring Boot Microservice]
|
Input Validation
|
[CRUD Operations (Local JSON File)]
|
Logging/Tracing
|
[Log/Trace Collector]
|
[GenAI Root Cause Analyzer]
|
[Issue Detection & Root Cause Report Generation]
|
[Local File Storage]
|
[Approval Workflow]
|
[Code Change Application (Self-Healing)]

|
[Response to Client]

Legend: Solid arrows indicate process flow; square brackets denote logical components[7][9].
Component breakdown:

1. **Client**: Issues POST requests to the service.
2. **Spring Boot Microservice**: Receives, validates, handles CRUD, triggers logging, and delegates to AI engine.
3. **CRUD (Local JSON Files)**: Replaces databases with flat-file persistence, ensuring portability and local-only operation[11].
4. **Logging/Tracing**: Captures operational metadata for diagnosis (with Spring Actuator, Sleuth, etc.).
5. **Log/Trace Collector**: Aggregates logs, traces, and error events.
6. **GenAI Root Cause Analyzer**: Consumes log data, performs anomaly/RCA, generates human-readable explanations and fixes[3][5].
7. **Issue Detection & Report**: Textual summary written to local storage.
8. **Approval Workflow**: Waits for manual or automated approval before enacting code/config changes.
9. **Code Application (Self-Healing)**: Applies fixes, updates configuration, or restarts subsystems locally.
10. **Client Response**: Communicates status, errors, or results back to requesting client.

The essence of this flow is full local operation, security by isolation, auditability by report logs, and extensibility for further integration with distributed or production-grade systems.

---

3. Core POC Architecture Components: Detailed Validation and Best Practices

3.1 Spring Boot Microservice: Request Handling & Validation

Request Pattern Requirements:

- POST endpoint(s) accepting JSON, mapped to domain model(s).
- Comprehensive validation, including:
  - Bean validation (JSR-380/JSR-303 annotations: @NotNull, @NotBlank, @Pattern, etc.)
  - Custom validators as needed (e.g., restricted values, cross-field conditions)[11][13].
- Exception handling that returns rich error responses (HTTP 400/422) while logging for later diagnostics.
- Support for integration of validation errors into the log/trace pipeline for downstream RCA by GenAI.

Validation Techniques:

@PostMapping("/entity")
public ResponseEntity<?> createEntity(@Valid @RequestBody Entity entity, BindingResult bindingResult) { ... }

- Custom exception handler using @ControllerAdvice to ensure all validation errors are consistently reported and optionally stored[11].

Elaboration:
Spring Boot validation combines annotation-based constraints and optional service-layer validation using the @Validated annotation. It is critical to ensure that the validation logic covers all input vectors and edge cases, logging both standard and custom errors for exhaustive RCA coverage. Unit and integration tests should exercise the validation logic for positive and negative cases[13].

3.2 Local JSON File-Based CRUD Operations

Pattern Requirements:

- No connection to external databases (RDBMS, NoSQL, or cloud services); all data must persist in local JSON files, preferably under a dedicated directory for see-through audit, portability, and ease of testing.
- Support standard CRUD operations conforming to RESTful principles (e.g., /entities [GET
- Thread-safe, lock-coordinated file access to prevent data corruption on concurrent writes.
- File backup or versioning strategy for post-repair audit and rollback if necessary[10][15].

Implementation Strategies:

- Use java.nio.file and/or Jackson/Gson library APIs for atomic read/write, with explicit file-lock coordination when necessary.
- For concurrent environments, leverage Java's FileLock on NIO channels, or simpler synchronized blocks for single-service processes.
- Consider a per-entity JSON file model (each record is a file), or a single collection file (with array-of-objects), depending on application scale and performance trade-offs.
- Ensure all operations are robust against partial writes (e.g., perform write to temp file and rename atomically).

Elaboration:
This approach is lightweight yet presents challenges when high concurrency arises or record sizes grow. Still, for POC or demo usage it offers maximum transparency and minimizes dependencies. Careful file access practices and exception/logging pipelines must be in place to guarantee data durability and traceability[17][18].

3.3 Logging and Tracing Collection

Pattern Requirements:

- Automatic capture of business, validation, and infrastructure errors and informational events via standard Java logging (e.g., SLF4J or Logback) and Spring's tracing facilities.
- Optional integration with OpenTelemetry or Spring Boot's native tracing/metrics APIs for detailed, structured logs and distributed trace compatibility (if future evolution is planned)[20].
- Use unique request or trace IDs for correlating logs relevant to a particular incident, so downstream AI/RCA models can reconstruct full context.

Best Practices:

- Use Spring Boot Actuator for health and diagnostics endpoints[21].
- Integrate structured logging (JSON output) to simplify AI model consumption and post-mortem analysis.
- Collect traces for POST and critical operations; for POC, traces can be written locally or, for demonstration, integrated with local Jaeger/Tempo instances as optional enhancements[20].

Elaboration:

Observability (logs, traces, metrics) forms the foundation for anomaly detection, RCA, and reliable failure handling. Ensuring that data is machine-parseable (preferably JSON) and includes correlation identifiers is fundamental to automated and AI-enabled analysis.

3.4 Lightweight GenAI Model Integration (RCA and Report Generation)

Pattern Requirements:

- GenAI model must run entirely locally on Mac hardware, supporting privacy and offline operation. Accepted choices include Docker Model Runner (for running Llama, Gemma, DeepSeek, etc.)[23], LocalAI via Docker[24], or Java wrappers around ONNX models or other LLMs using Spring AI or LangChain4J[24].
- Integration must allow the microservice to invoke the model to:
  - Summarize errors, logs, and traces as context-rich natural language.
  - Perform anomaly/root cause analysis using prior cases, template matching, or outlier/similarity detection[3][25].
  - Suggest remediations (with embedded controls limiting code modification to "explanation only" pending approval).
  - Generate a Markdown, plaintext, or JSON structured report that is stored locally.

Implementation Steps:

- Wrap AI model as a local REST service (via Docker) or embed as a Java class with JNI or cross-language bridges as needed.
- Use HTTP calls or direct library calls from Spring Boot to process log files or traces and receive summarized diagnostics.
- For OpenAI-compatible endpoints, tools such as Spring AI support native integration (see configuration scripts in live repos)[24].

Elaboration:
The right AI model should be low-latency, explainable, and trainable (e.g., fine-tuned on local log/error corpora for better precision). Hosting the model via Docker-whether Model Runner or LocalAI-ensures that all processing is physically isolated and data never leaves the device, crucial for privacy and regulatory compliance demonstrations[24].

3.5 AI-Driven Anomaly Detection and Root Cause Analysis

Pattern Requirements:

- The AI system must analyze logs/traces for:
  - Known error patterns (rule or prompt-based LLM querying).
  - Anomalies (distribution shifts, outlier detection, context correlations).
  - Cross-system correlation (if multiple services are in scope; for POC, may remain single-service).
- Use NLP and time-series AI techniques (e.g., word2vec, prompt engineering, outlier scoring) to turn raw entries into structured insights and RCA hypotheses[26][5].
- AI-generated recommendations should be clear, actionable, and reviewable by a user/human.

Implementation Suggestions:

- Continuously feed error logs to the GenAI pipeline.
- Trigger in-depth RCA generation on detected anomalies or after a configurable error rate threshold is reached.
- Leverage AI models with an audit trail: include the log snippet, time, and any relevant code or configuration details in the output report for transparency.

Elaboration:
Current state-of-the-art systems achieve high-fidelity RCA using a blend of anomaly detection (e.g., unsupervised clustering, metric baselining) and LLM-based summarization for human consumption. Incorporating feedback loops (tagging reports as correct/incorrect) allows on-the-fly refinement of model recommendations, ensuring continuous improvement even in local deployments[4].

3.6 Text Report Generation, Local Storage, and Retrieval

Pattern Requirements:

- Generated RCA and fix reports (from GenAI) should be stored in human-readable, versioned files (e.g., Markdown, plaintext, or JSON), indexed by timestamp and incident/correlation ID.
- Text files provide traceability, transparency, and offline sharing for audits or support escalation (all critical for acceptance testing).

Implementation Details:

- Use Java IO, NIO, or high-level wrappers (e.g., Jackson for JSON) to write reports atomically and reliably[17][18].
- Ensure that:
  - File writes do not overwrite or truncate prior reports unless explicitly intended.
  - Retrieval endpoints are available for authorized users to review the reports before any code change is applied.

Elaboration:
Design decision to keep all reports locally ensures privacy but requires robust permissions and file management practices to prevent unauthorized access or accidental deletion. For advanced scenarios, add checksums or digital signatures to reports for tamper-evidence.

3.7 Approval Workflow Integration

Pattern Requirements:

- Code or configuration changes, especially those that touch business logic, must be gated by an explicit human (manual) approval process within the application.
- Integration of a lightweight approval workflow inspired by proven patterns described in Spring-based human workflow engines or simple approval state machines[28][29].
- When approval is granted (via UI, API call, or CLI command), the system triggers the GenAI model or internal script to safely apply the fix and logs the outcome.

Implementation Techniques:

- Approval state should be atomic and resilient across restarts (maintain workflow state in a JSON audit log).
- Consideration for session persistence and user authentication if workflow is exposed beyond localhost (see security below).
- Provide status and audit endpoints to retrieve pending, approved, or rejected changes.

Elaboration:
The approval flow not only serves as a control point for safety and audit, but it also facilitates testing and demonstration of "human-in-the-loop" interventions-a critical requirement for enterprise system trust and regulatory compliance.

3.8 Automated Application of Fixes

Pattern Requirements:

- Once a fix is approved, the system should:
  - Safely enact the change (hot-reload configuration, patch code, or restart a subsystem as appropriate).
  - Log both the attempted change and the result (success, failure, rollback).

    - Keep the change atomic and reversible where feasible: e.g, for patched configuration, backup old version before overwrite.
- For Spring Boot, leverage dynamic configuration reload patterns or restart the service as appropriate.
- For session or runtime changes, ensure the system persists service state and user sessions across updates where needed (see session management below)[31][32].

Implementation Insights:

- Use Spring Boot's graceful shutdown and config reload mechanisms (see Actuator, @RefreshScope, and Spring Cloud Config patterns) to minimize downtime and user disruption[34][35].
- For configuration or code changes, scripts or shell commands invoked by the microservice can alter local files and signal the app or container to reload/restart.

Elaboration:
This is the step with the highest risk for system disruption or recurrence of failure if not handled atomically, so the rollback facility and comprehensive post-action logging are crucial for reliability and trust.

---

4. Expansion and Enhanced Patterns for Robustness

4.1 Dynamic Configuration Reload and Hot Swapping

A robust self-healing system POC is greatly enhanced by supporting configuration hot-swapping. This is both safer and faster than restarting the entire service for small-to-medium configuration changes. Patterns to accomplish this include:

- Using @RefreshScope and actuator /refresh endpoints for live config reloads in Spring Boot (backed by a local config file)[33].
- Watching configuration files for changes and signaling the application via in-memory event or endpoint to reload affected beans.
- Pluggable override: in case of unrecoverable error, fallback to default configurations or activate circuit breakers to protect the system.

Attention:
Mishandling dynamic config can cause partial application or race conditions-ensure reloads are atomic, consistently logged, and, where possible, validated before application.

4.2 Observability and Distributed Tracing

While POC may be single-service, building in support for distributed tracing with OpenTelemetry primes the stack for production scenarios. Recommendations:

- Use the OpenTelemetry Java SDK and micrometer-tracing-bridge-otel for seamless metric, trace, and log exports, even if consumed locally via Jaeger/Grafana Tempo
- Instrument all asynchronous and major service interactions for traceability
- Propagate trace and request IDs in all logs and reports, binding root-cause analyses to the precisely affected request chain[20].

4.3 GenAI Model Security and Safe Execution

Running AI models with the ability to read, write, or patch system code/config introduces new security risks, even when purely local. To mitigate:

- Run the AI process under a separate, limited-privilege system account, restricting access to only required files and sockets.
- Use an explicit allow-list of files/directories the AI model may read or write.
- Sanitize all AI-proposed code or config changes, scanning for malicious or unsafe instructions before application.
- If any networking is enabled for code or model updates, restrict endpoints strictly to local loopback.

4.4 Session Persistence and Concurrency in Local Operation

For approval workflows and multi-step logical flows, ensure session data is persistent across restarts (by enabling Spring Boot's server.servlet.session.persistent=true and, where applicable, a file-, JDBC-, or in-memory-store-backed session engine)[32]. For CRUD file access, use OS-level file locks or database-like transaction ID patterns to prevent simultaneous conflicting writes.

4.5 File Storage Atomicity, Recovery, and Audit

- Always backup files prior to mutation; offer users ability to restore from backup if self-healing fix is rejected or fails.
- Store all AI-generated reports, approval actions, and post-fix logs in an immutable audit log (append-only preferred).
- For sensitive file operations, validate file integrity (using checksums, digital signatures, or hashes) before accepting reports or actions for further assurance.

4.6 Advanced: Plug-in or Modular AI Analysis

If more advanced RCA and anomaly detection are required, adopt a plug-in architecture, where multiple AI or ML algorithms (rules, clustering, LLM summarization) can be swapped or

combined in the analysis workflow. The modularization also allows external researchers or engineers to contribute new AI models without risking core service stability.

---

5. Security, Privacy, and Access Control

All self-healing and AI-augmented remediation must balance automation with security:

- **Restrict approval endpoints** to localhost or secured authentication.
- **Sensitive config and logs** storing PII or credentials should be encrypted at rest or redacted before being passed to GenAI processes or included in reports.
- **All code change operations** should run in a sandbox or restricted context; under no circumstances should the model be allowed to access arbitrary files, invoke system commands, or carry out unapproved network actions.
- **Monitor/alert** on all fix attempts, population of new approval requests, and actual approval/application events for audit and anomaly detection in the healing process itself.

---

6. Architectural Diagram and Documentation Conventions

6.1 Text-Based Architecture Diagrams

Text-based (ASCII) architecture diagrams:

- Favor clarity, conciseness, and explainability over exhaustive detail.
- Should be present in situ with the code (e.g., as code comments), maintained alongside logic so as not to become stale[36].
- Serve not only as static documentation but as orientation points for onboarding, debugging, and future editing.

Example:

```
[ Client ] ---> [ REST POST Request ]
|
v
[Spring Boot Microservice]
|
v
[Validation & JSON CRUD]
|
+----> [Log/Trace Collector] --> [GenAI Root Cause Engine]
|
v
[Textual RCA / Fix Report (local file)]
|
```

[Approval Status: pending | approved]
|
v
[Apply Fix (Local Patch/Reload)]

Each block in the diagram corresponds to a tangible code module, script, or subsystem, and is annotated or hyperlinked to corresponding source code or documentation as recommended by text-based documentation best practices[37].

---

7. Comparative Table: Critical Self-Healing Patterns for POC

Pattern	Description	POC Implementation	Recommended for POC?	Notes
Circuit Breaker	Halt failing service calls temporarily	Not required (single-service, local)	Optional	Spring Cloud compatible
Bulkhead	Resource isolation per service/component	N/A for single-service	No	Advanced use
Graceful Shutdown/Restart	Shutdown service only after in-flight requests complete	Spring Boot, actuator	Yes	Use server.shutdown: graceful
Approval Workflow	Gate change approval	Custom state machine, JSON persistence	Yes	Session persistence required
Log/Trace to RCA	Automated GenAI-powered RCA on error	Local LLM + report Gen	Yes	Ensure comprehensive logs
Config Reload	Hot-swap configuration post-fix	@RefreshScope, actuator	Yes	Must validate reload
File Locking	Prevent concurrent corrupt writes	NIO file locks, synchronized blocks	Yes	Critical for multi-threaded ops
Text-based Audit Logs	Trace all actions, RCA, approvals/fixes	Local CSV/JSON/Markdown	Yes	Immutable, append-only files

---

8. Industry Alignment and Further Recommendations

Self-Healing Patterns in Production: Alignment

- **Health Checks + AI Remix:**
  Industry best practices use Spring Actuator for "liveness" and "readiness" probes, combined with user- or AI-driven health metric analysis. While full automation is risky in prod, POCs should demonstrate the entire self-healing control loop, with clear demarcation of points requiring human sign-off[2].
- **ModelOps for GenAI:**
  Local models are rapidly catching up to SaaS LLMs in core RCA tasks (summarization, anomaly detection, fix suggestion). Docker Model Runner, LocalAI, and LangChain4J are all viable for rapid POC development on Mac hardware[24][23].
- **Observability & Security:**
  Modern observability stacks (e.g., ELK, OpenTelemetry, Prometheus) add enormous depth; for POC, logging to JSON + tracing with On-the-fly metrics is sufficient, but foundation for later ops migration should be present[20][7].

Conclusion: Assessment and Readiness for Self-Healing POC

The proposed and reviewed proof-of-concept for a self-healing Spring Boot microservice system-with full file-based local persistence, AI-driven log analysis, and safe approval workflows-is fundamentally sound, alignable with industry practice, and suitable for local demo on modern Mac hardware. The core workflow of request-validation → CRUD → logging → AI RCA → report → approval → fix, encapsulated in the architecture, is robust and extendable.

Critical success factors and next steps:

- All data operations, especially those involving logs, models, and fixes, must be atomic, auditable, and secure.
- The GenAI component should remain local, privacy-preserving, and limited to explanatory and suggestive roles unless an explicit risk-managed sandbox is in place for code changes.
- Approval workflow must be explicit, logged, and session-resilient across restarts.
- Observability should be structured (JSON logs, correlation IDs), facilitating automated AI analysis and manual troubleshooting.
- The system should be built incrementally, with all AI-driven actions (especially code changes) gated behind extensive integration and regression tests.
- Documentation-especially the architectural ASCII diagram-must remain up-to-date, in situ, and directly connected to logic to maximize onboarding and troubleshooting value.

Final verdict:

With the refinements and patterns outlined above, the POC architecture is correct, safe, extensible, and ready for robust self-healing system development-demonstrating a seamless

fusion of resilient Spring Boot microservices and next-gen local GenAI diagnostics and remediation.

References (37)

1. *Self-Healing Systems - System Design - GeeksforGeeks*. https://www.geeksforgeeks.org/system-design/self-healing-systems-system-design/
2. *Self-Healing Microservices: Implementing Health Checks with Spring Boot ...*. https://www.javacodegeeks.com/2025/07/self-healing-microservices-implementing-health-checks-with-spring-boot-and-kubernetes.html
3. *Run GenAI Models Locally with Docker Model Runner*. https://dev.to/docker/run-genai-models-locally-with-docker-model-runner-5elb
4. *Spring Boot Integration + LocalAI: Code Conversion - DZone*. https://dzone.com/articles/spring-boot-integration-with-localai-for-code-conversion
5. *Tutorial: Log Anomaly Detection Using LogAI*. https://opensource.salesforce.com/logai/latest/tutorial.build_log_anomaly_detection_app.html
6. *Top 8 AI-Powered Anomaly Detection Tools for Time Series Data*. https://www.anodot.com/learning-center/top-8-ai-powered-anomaly-detection-tools-for-time-series-data/
7. *OpenTelemetry Java Microservices: The Complete Guide to Distributed ...*. https://www.springfuse.com/opentelemetry-java-microservices/
8. *Creating simple workflow engine with database and JAVA*. https://stackoverflow.com/questions/46722376/creating-simple-workflow-engine-with-database-and-java
9. *java - Activiti workflow, the approval system is easily implemented ...*. https://segmentfault.com/a/1190000040625094/en
10. *Session Management Migrations :: Spring Security*. https://docs.spring.io/spring-security/reference/migration/servlet/session-management.html
11. *Spring Boot Session Persistence - Runebook.dev*. https://runebook.dev/en/articles/spring_boot/application-properties/application-properties.server.server.servlet.session.persistent
12. *Reloading Properties Files in Spring - Baeldung*. https://www.baeldung.com/spring-reloading-properties
13. *Reloadable properties file with Spring using Apache Commons ...*. https://examples.javacodegeeks.com/java-development/enterprise-java/spring/reloadedable-properties-file-spring-using-apache-commons-configuration/
14. *How to Make Dynamic Configuration Changes in Spring Boot Without ...*. https://dev.to/adityabhuyan/how-to-make-dynamic-configuration-changes-in-spring-boot-without-restarting-your-application-1f7f
15. *ASCII Architectures - GitHub*. https://github.com/prashikdewtale10/ascii-architectures

16. *How to Build a Self-Healing Java Microservices Architecture*. https://www.springfuse.com/self-healing-microservices-architecture-in-java/
17. *Harnessing Generative AI for Root Cause Analysis - SAP Savvy*. https://sapsavvy.com/harnessing-generative-ai-for-root-cause-analysis/
18. *Scenario and Root Cause Analysis with Generative AI - Coursera*. https://www.coursera.org/learn/scenario-and-root-cause-analysis-with-generative-ai
19. *Taking ASCII Drawings Seriously: How Programmers Diagram Code*. https://dl.acm.org/doi/fullHtml/10.1145/3613904.3642683
20. *Spring REST Validation Example - Mkyong.com*. https://mkyong.com/spring-boot/spring-rest-validation-example/
21. *The Power of AI in Root Cause Analysis - EasyRCA*. https://easyrca.com/blog/the-power-of-ai-in-root-cause-analysis/
22. *Validating RequestParams and PathVariables in Spring*. https://www.baeldung.com/spring-validate-requestparam-pathvariable
23. *Spring Restful Web Services JSON CRUD Example - Dinesh on Java*. https://www.dineshonjava.com/spring-restful-web-services-json-crud-example/
24. *Spring Boot - Consuming and Producing JSON - GeeksforGeeks*. https://www.geeksforgeeks.org/advance-java/spring-boot-consuming-and-producing-json/
25. *Essential Principles of Spring Boot Microservices Architecture*. https://mobisoftinfotech.com/resources/blog/essential-principles-spring-boot-microservices
26. *Reading and Writing Text Files in Java - Build With Sachin Blogs*. https://blog.sachingurjar.me/reading-and-writing-text-files-in-java/
27. *A Comprehensive Guide to Reading and Writing Files in Java*. https://codingtechroom.com/tutorial/java-reading-and-writing-files-in-java
28. *Documenting System Architecture With AsciiDoctor*. https://dojofive.com/blog/document-system-architecture-ascii-doctor/