Module 7 Capstone Milestone: Software Configuration Management

Alexander Ricciardi

Colorado State University Global

CSC480: Capstone Computer Science

Dr. Shaher Daoud

July 27, 2025

Module 7 Capstone Milestone: Software Configuration Management

The Mining Regulatory Compliance Assistant (MRCA) is an AI-powered web application based on microservices architecture and a novel Advanced Parallel HybridRAG (APH) system. MRCA allows users to query Mine Safety and Health Administration (MSHA) regulations using natural language. This document provides an overview of the MRCA Software Configuration Management (SCM) process. An SCM is a configuration management process that supports, throughout a software life cycle, the software's management, development, maintenance, quality, and users (Washizaki, 2024). The MRCA's SCM methodology is tailored for a solo-developer team and addresses the unique MRCA needs, such as handling LLM integrations, graph databases with vector embedding, a novel RAG system, and frequent iterations within an 8-week development period. The SCM process utilizes Git with GitHub for version control, incorporating integrated branching strategies, change control, and testing.

MRCA's SCM Methodology

MRCA's SCM methodology is based on IEEE 828-2012 standards modified for a small-scale, AI-powered project. IEEE 828-2012 is a standard designed and maintained by the Institute of Electrical and Electronics Engineers (IEEE) for Configuration Management in Systems and Software Engineering (IEEE, 2012). The standard describes how the configuration management processes need to be established, implemented, who the responsible parties are for doing specific activities, when these activities are to happen, and what specific resources are required (NASA, 2022). The IEEE 828-2012 standards, modified to accommodate the MRCA-specific RAG/AI requirements, that is, include configuration items not typically found in traditional software, such as datasets, LLM prompts, and AI model versions.

The MRCA's SCM methodology includes four core activities, which are configuration identification, change control, status accounting, and audit/testing. Configuration identification defines the code base structure, documentation, and artifacts. This includes Configuration items (CIs), which are Python source files such as `backend/main.py` and `frontend/bot.py`, data scripts stored in standalone files (e.g., `build_data/cfr_downloader.py)`, library dependencies (`requirements.txt`), and documentation (e.g., UML diagrams).

Change Control Configuration

Change control uses a Git version control system that is managed through GitHub with GitFlow-inspired branching that integrates a GitFlow-Lite branching strategy. GitFlow is a branching model that uses multiple branches to transfer code from development to production (AWS, n.d.). GitFlow-Lite is a subset of Git-Flow, with a simplified branching structure that incorporates a long-lived Development trunk branch and short-lived Feature branches (Syntevo, n.d.). The MRCA's SCM branching model integrates the GitFlow-Lite by implementing a `main` branch for production-ready code, a `develop` branch for code implementation, and feature branches such as `feature/parallel-hybrid` for implementing and testing new features, and a `hotfix` branch for urgent fixes. This SCM Git version control system is well-suited for a solo-developer team with an 8-week development timeline by reducing the complexity and overhead of a full GitFlow model.

Status Accounting System and Audit Configuration

The status accounting system tracks versions using a versioning naming structure (version 2.0.0) for major component releases and new feature implementation, like implementing a core new APH component, and for hotfix and feature improvement (e.g., version 2.0.1); note that releases are Git merges into the `main` branch from the `develop` branch or a `hotfix` branch.

In addition to being documented in the project development documentation, these GitHub Releases are documented using `git commit -m"…"` and accessible using `git log`. Audits are set bi-weekly through GitHub Actions workflows that run tests automatically and check coverage (>80%), using tools like Dependabot (GitHub, n.d.a) to monitor and test dependencies for vulnerabilities. Note that audits are security and integrity code automated checks implemented using GitHub Actions workflows after the code has been merged (GitHub, n.d.a). In the context of MRCA's SCM process, the audit is executed after a merger is applied to the `main` branch. These status accounting and audit approaches provide a verifiable history of the project, ensuring that every code change is both documented and audited for security and integrity.

Seamless Evolution: Change Control Process

For MRCA, the processes Software Change Request (SCR) and Change Control Board (CCB) are integrated within the Git/GitHub workflow, enabling the development process to proceed without interruptions by following an iterative steps process based on a Kanban-Agile development methodology. The step can be defined as follows. Step 1: create a GitHub Issue describing the modification, for example, "adding circuit breaker in backend/circuit_breaker.py". Step 2: create a feature branch from develop, for example, `git checkout -b feature/circuit-breaker`, then implement/commit the changes by using git commit `git commit -m "Add CircuitBreaker class"`. Step 3: design and run unit tests, for example, by performing a `pytest tests/unit/test_circuit_breaker.py` and by performing integration tests. Note that testing is executed directly on the `feature/circuit-breaker` branch. Step 4: perform a Pull Request (PR), do a self-review, and `feature/circuit-breaker` branch is merged into the `develop` branch after ensuring that the changes pass the CI checks. Step 5: merge `develop` branch into the `main` branch with a tag release (e.g., git tag v2.1.0), and push to

GitHub. Step 6: update the project documentation and the changelog, and run audits via workflows.

MRCA's SMC Methodology Function and Justification

This seamless SMC process supports MRCA's Kanban-style iterations, allowing merges to be reverted using `git revert`, respecting the development timelines, developing, implementing, testing, monitoring, fixing bugs, and auditing code for security and integrity issues. This audit process makes MRCA a secure, robust, and maintainable web application. Moreover**,** by adopting a GitFlow-Lite within GitHub distributed version control, the SMC process allows for offline work, branching for parallel development, and traceable code changes via commit history. Including benefits such as monitoring changes for troubleshooting using `git diff` and assurance of quick recovery from AI, database, and APIs integration issues, like reverting faulty LLM prompts in `hybrid_templates.py` or rolling back a data processing script that corrupted the Neo4j graph. This minimizes the risks of deployment failures and data corruption. In other words, the SMC process is well suited for a solo-development team by implementing automated workflows, lite branching, and reducing manual overhead significantly, while maintaining high-quality standards for software development and implementation that is secure and minimizes the risk of project failure due to technical debt (unreliable, easy, quick-fix solutions) or unclear structure or plan as more features are added.

Testing Plan

The testing plan, integrated into SCM, includes unit, integration, end-to-end (E2E), reliability, and architecture tests. These tests are executed using test cases based on MRCA's Architecturally Significant Requirements (ARS) (Keeling, 2017). Below is a list of the test cases used to measure the quality of the MRCA system and its microservices/APH architecture.

Table 1

Test Cases and SCM Controls

Test and Risk Scenario	SCM Control	Metrics	Benefit / Justification
Test case 1 - Regulatory Citation Retrieval - Component Integration Test Faulty vector embedding and graph retrieval	Implement `feature/*` branch control and control the configuration of the retrieval process. A failure will be a malformed citation and a MASH regulation section missing	≥ 95 % passage retention; p95 latency < 120 ms	This test verifies how the integrated components of the system work together (Das, 2024). It verifies the quality of the retrieval process based on the AHP system
Test case 2 - Multi-Domains Query - Component Integration Test Faulty vector embedding and graph query results, and intelligent fusion (`hybrid_fusion`).	Implements PR checklist, latency, and integration validity. Controls the configuration of the fusion validity across multiple MSHA regulatory domains.	Graph DB latency < 150 ms `fusion_quality` score ≥ .70	Prevents runtime failures; same rationale as test case 1. Test if the integrated components of the system fail when they are working together, and measure the quality of the fusion.
Test case 3 - Reliability Under Degraded External Services - End-to-End Testing API's faulty communication that may cause cascading faults	Implements circuit-breaker and hot-hot-fix branch control configurations. Test uses a chaos testing approach. It tests the circuit breaker functionality, on/off states, retry/backoff, and user messaging when the Neo4j database, LLM providers, or network links or API fail	Graceful 503; breaker opens ≤ 5 fails.	Helps to improve system resilience in microservices architecture by using circuit breakers to control issues related to network instability and unresponsive services (Krishna, 2023).
Test case 4 - Reliability, Fault-Injection Test, and Unit Tests Latency under heavy load	Control CI. The test uses chaos scenario fault control. It tests the performance of the system under concurrent load by injecting faults	p95 < 1.2 s; success ≥ 99 % LLM's response time should be 10–35s	Chaos testing helps to identify latent faults, improving the system's ability to respond to overcrowding, network delay, and service outages (Chintale et al., 2023).
Test case 5 - Confidence Score – Overall Architecture Evaluation (AHP) Hallucinations, off-topic prompts	Implements commit history control configuration by using `git revert`. It uses Hallucination tests by inputting 50 off-topic prompts. It tests if the system detects unsupported prompts and checks response confidence outputs	OOS precision ≥ 95 %; hallucination ≤ 5 % ; rollback ≤ 5 min	Allows for rapid recovery from hallucinations and aligns with HaluBench benchmark protocol for handling hallucinations (Ravi et al., 2024).
Test case 6 - Dependency Vulnerabilities and Security – Audits	Implements audit configuration control. The Dependabot tool is used to block vulnerable libraries (Dependencies).	Dependabot reports a vulnerable dependency	Main branch merge protection on reported dependency vulnerability. Allows for secure production code delivery. Protects production code.

Note: The table describes the test cases, their related SCM control, expected test output metric, and their benefits/justification.

Conclusion

The MRCA's SCM process is a configuration management methodology that is well-suited for addressing the software management challenges posed by MRCA's AI-powered and

novel APH system. It implements a GitFlow-Lite branching approach, an automated workflow, and a testing plan. It also caters to a solo-development team, and it provides traceability, stability, and seamless evolution of the application's architectural components. Additionally, the SMC testing plan strategy mitigates risks, manages complexity, and helps to successfully deliver a high-quality and maintainable application.

References

AWS. (n.d.). Gitflow branching strategy. *Choosing a Git branching strategy for multi-account DevOps environments*. AWS Documentation. https://docs.aws.amazon.com/prescriptive-guidance/latest/choosing-git-branch-approach/gitflow-branching-strategy.html docs.aws.amazon.com

Chintale, P., Pandiyan, A., Chaudhari, M., Chigurupati, M., Desaboyina, G., & Malviya, R. K. (2023). *Serverless chaos engineering: A framework for fault injection and resiliency testing in AI-powered cloud workflows*. Journal of Harbin Engineering University, *44*(12), 1577-1584. https://harbinengineeringjournal.com/index.php/journal/article/view/3460

Das, S. (2024, November 4). *Integration testing and unit testing in the age of AI*. Aviator Blog. https://www.aviator.co/blog/integration-testing-and-unit-testing-in-the-age-of-ai/

GitHub. (n.d.a). Keeping your supply chain secure with Dependabot. *Securing code*.GitHub Docs. https://docs.github.com/en/code-security/dependabot

GitHub. (n.d.b). Auditing security alerts. *Securing code*. GitHub Docs. https://docs.github.com/en/code-security/getting-started/auditing-security-alerts

IEEE. (2012). *IEEE standard for configuration management in systems and software engineering* (IEEE Std 828-2012). https://doi.org/10.1109/IEEESTD.2012.6170935

Keeling, M. (2017). Chapter 12: Give the architecture a report Card. Design it! From programmer to software architect. Pragmatic Bookshelf. ISBN-13: 978-1-680-50209-1

Krishna, H. (2023, October 3). *What is circuit breaker in microservices?* SayOne Tech Blog. https://www.sayonetech.com/blog/circuit-breaker-in-microservices/

NASA. (2022). Chapter 5: Software Configuration Management. *NASA procedural requirements for software engineering (NPR 7150.2D* [Directive]. NASA Office of the Chief Engineer. https://nodis3.gsfc.nasa.gov/displayDir.cfm?Internal_ID=N_PR_7150_002D_&page_name=Chapter5&utm_source

Ravi, S. S., Mielczarek, B., Kannappan, A., Kiela, D., & Qian, R. (2024, July 11). *Lynx: An open-source hallucination evaluation model* (arXiv:2407.08488) [Preprint]. arXiv. https://doi.org/10.48550/arXiv.2407.08488

Syntevo (n.d.). Git-Flow Light. *SmartGit manual: Development processes*. Syntevo Docs. https://docs.syntevo.com/SmartGit/Latest/Manual/DevelopmentProcesses/Git-Flow-Light

Washizaki, H., (Eds.). (2024). Chapter 8: Software Configuration Management. *Guide to the Software Engineering Body of Knowledge (SWEBOK guide)* (Version 4.0). IEEE Computer Society. https://www.swebok.org