Modern Topics

Machine Learning, Cybersecurity,
Hyperautomation

Machine Learning

Analysis

- Requirements are **probabilistic**.
- In traditional systems, requirements are stated as "Given X, the system shall output Y," whereas in ML systems, they are expressed as "Given X, the system should output Y with at least 95% confidence."
- This makes requirements specification, validation, and testing more complex. So, treat as **defining performance goals**, not fixed behaviors.

Analysis

- Implication:
  - ✔ ML requirements include accuracy, precision, recall, F1-score, fairness, and latency.
  - ✔ Must account for uncertainty, edge cases, and model drift.
  - ✔ Functional requirements are now accompanied by data requirements:

Design

- An ML system isn't just "a model" — it's a **pipeline of components**.
- This means:
  - ✔ Design for versioning, modularity, and reproducibility.
  - ✔ Data lineage and feature provenance become critical.
  - ✔ Training code is as important as inference code.

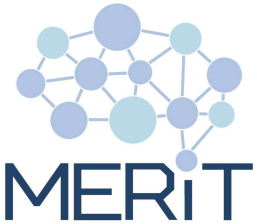
Design

- Implication:
  - ✔ Need to **architect pipelines** using modular, testable, and swappable components.
  - ✔ Use patterns like **Model Registry**, **Feature Store**, and **Retraining Scheduler**.
  - ✔ Support for **A/B testing**, **shadow mode**, and **canary rollouts** should be planned.

Testing

- Traditional testing focuses on deterministic logic and expected outputs.
- ML testing must evaluate:
  - ✔ Model performance on real-world data
  - ✔ Bias and fairness
  - ✔ Robustness to adversarial inputs or corrupted data
  - ✔ Data quality, feature leakage, and label noise

Testing



- Test Types Unique to ML:
  - ✔ Data tests: Schema, distribution, completeness, drift detection
  - ✔ Model tests: Accuracy, ROC curves, calibration
  - ✔ Integration tests: Model + pre/post-processing + service
  - ✔ Bias/fairness tests: Demographic parity, disparate impact

Deployment

- Unlike static code, ML models decay over time (data drift, concept drift).
- Implications:
  - ✔ You must monitor not just uptime, but:
    - ❖ Model performance in production
    - ❖ Input/output distribution drift
    - ❖ User feedback loops

Deployment

- Implications:
  - ✔ Need for automated retraining pipelines or human-in-the-loop review.
  - ✔ Deployment artifacts now include:
    - ❖ Trained model binaries
    - ❖ Feature transformers
    - ❖ Training config + random seed + dataset version

Maintenance

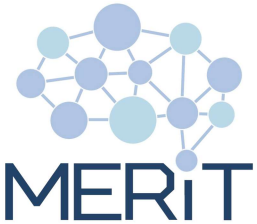
- Traditional SDLC ends with “maintenance = bug fixes.”
- ML SDLC includes ongoing learning cycles:
  - ✔ Collect data → retrain → evaluate → redeploy

Maintenance

- Implications:
  - ✔ Must plan for continual labeling, feedback ingestion, and model governance.
  - ✔ Maintenance includes bias audits, compliance checks, dataset drift detection, and model retirement plans.
  - ✔ MLOps tooling becomes essential (e.g., MLflow, TFX, Kubeflow, SageMaker).

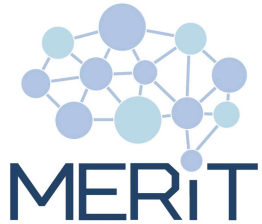
Cybersecurity

Security



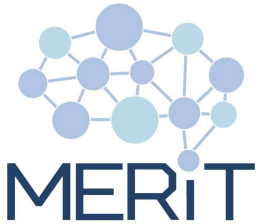
- Security is one of the **most critical** cross-cutting concerns.
- It spans the **entire SDLC** and must be addressed proactively, not reactively.
- A system that is functionally perfect but **vulnerable to attack** is **unusable** in practice.

Security in SDLC



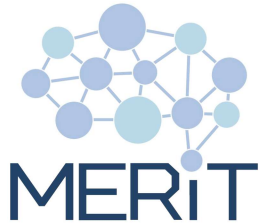
- Analysis: Identify security requirements such as authentication, authorization, encryption, logging, and compliance (e.g., GDPR, HIPAA).
- Design: Apply secure design principles (e.g., least privilege, fail-safe defaults), design threat models, and select secure architectures (e.g., zero-trust, separation of concerns).

Security in SDLC



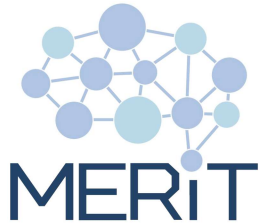
- Implementation: Avoid vulnerable coding practices (e.g., input sanitization, SQL injection), use static analysis tools, and enforce secure coding standards (e.g., OWASP).
- Testing: Conduct security testing (e.g., fuzzing, penetration tests, SAST/DAST), validate access controls and data protection, and simulate attacks.

Security in SDLC



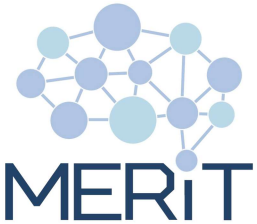
- Deployment: Secure configurations, HTTPS enforcement, secrets management, firewall rules, and container hardening.
- Maintenance: Monitor logs for anomalies, patch vulnerabilities, update libraries regularly, and respond to new vulnerability disclosures.

Threat Modeling



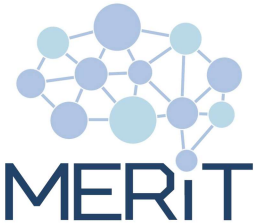
- A proactive method to identify potential security threats **before implementation**.
- Use models like **STRIDE** (Spoofing, Tampering, Repudiation, Information Disclosure, Denial of Service, Elevation of Privilege) to map threats systematically.
- It's like "designing for failure" but from the attacker's point of view.

Secure Design Principles



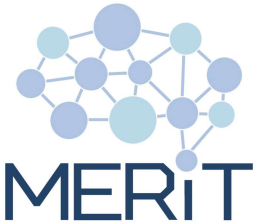
- **Principle of Least Privilege**: Users and components should have the minimum permissions necessary.
- **Fail-Safe Defaults**: Configurations should deny access by default, rather than allowing it.
- **Separation of Duties**: Avoid giving any single role too much control.
- **Defense in Depth**: Use multiple layers of security controls.

Secure Design Principles



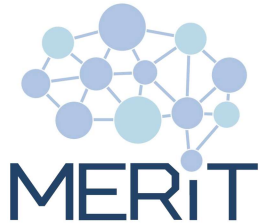
- **Don’t Trust User Input**: Assume all external input is malicious unless proven otherwise.

DevSecOps



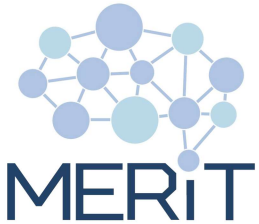
- Development + Security + Operations
- It is the practice of automating, integrating, and embedding **security into every phase** of the SDLC.
- Traditional SDLC: “Build first, test security later”
- DevSecOps: “Secure as you build and deploy continuously”

DevSecOps



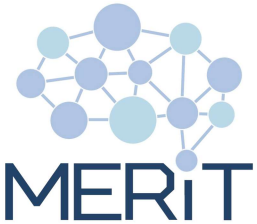
- Examples:
  - ✔ Run security scans on every commit
  - ✔ Include linting rules for secure coding
  - ✔ Block deploys with known vulnerabilities (e.g., in NPM or PyPI packages)

Security Testing Techniques



- **SAST (Static Application Security Testing)**: Analyze source code without running it.
- **DAST (Dynamic Application Security Testing)**: Scan the running system as a black-box attacker would.
- **Fuzzing**: Feed malformed or random input to crash or confuse the system.
- **Penetration Testing**: Manual or automated simulated attacks to uncover vulnerabilities.

Regulatory Compliance



- **GDPR (EU)**: Personal data collection, retention, and erasure rights.
- **HIPAA (US Health)**: Data protection in healthcare apps.
- **PCI DSS (Finance)**: Credit card processing security standards.
- **ISO/IEC 27001**: Information security management systems (ISMS).

Hyperautomation

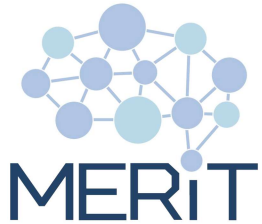
Definition

- Hyperautomation refers to the combined use of **AI, ML, RPA (Robotic Process Automation), and DevOps pipelines** to **automate** as many aspects of software engineering as possible.
- It is automating not just the application itself, but also the development of the application.

Definition

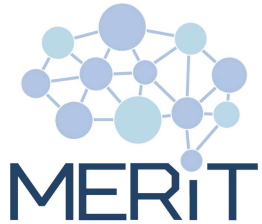
- Gartner defines hyperautomation as: “A disciplined, business-driven approach to rapidly identify, vet, and automate as many processes as possible.”

Manifest



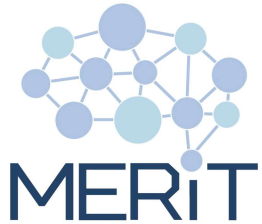
- It manifests as:
  - ✔ AI writing code,
  - ✔ Tools auto-refactoring or generating tests,
  - ✔ Bots analyzing PRs,
  - ✔ Smart pipelines triggering actions based on ML insights.

AI-Assisted Development



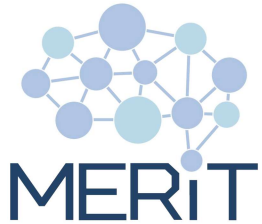
- Autocomplete code (e.g., GitHub Copilot)
- Suggest refactors (e.g., IntelliCode, Codota)
- Generate unit tests (e.g., Diffblue, TestPilot)
- Write docstrings and comments
- Detect vulnerabilities or code smells automatically
- Infer business logic from data

AI in SDLC



- Analysis: NLP models convert user stories into backlog items and generate acceptance criteria.
- Design: AI-based architecture analysis, compliance checkers, and auto-documentation from diagrams.
- Implementation: AI pair programming, boilerplate generation, and static code fix suggestions.
- Testing: Auto-generation of unit tests, visual regression detection, and smart test case prioritization.

AI in SDLC



- Deployment: AI-driven CI/CD orchestration (e.g., choose rollback vs. canary), anomaly detection.
- Maintenance: Predictive issue detection, automatic log analysis, self-healing infrastructure.

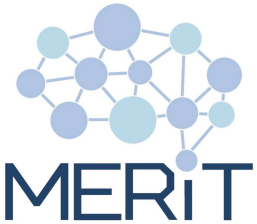
Risks and Challenges

- Hallucination & overconfidence: AI tools may generate incorrect or insecure code that looks plausible.
- Loss of control: Developers may rely too much on suggestions without a deep understanding.
- Bias and ethical issues: AI trained on biased codebases may replicate bad practices or unsafe assumptions.
- Security & compliance: Generated code might violate licensing, data handling, or security policies.

Risks and Challenges

- Explainability: It’s often unclear why the AI made a suggestion.

Best Practices



- Use AI as an augmentative, not an autonomous tool.
- Review AI-generated code like you'd review junior developer code.
- Train developers to understand the strengths and limitations of generative models
- Fine-tune tools to reflect your team's patterns and domain

Best Practices

- Combine with manual reviews, formal testing, and traceability.