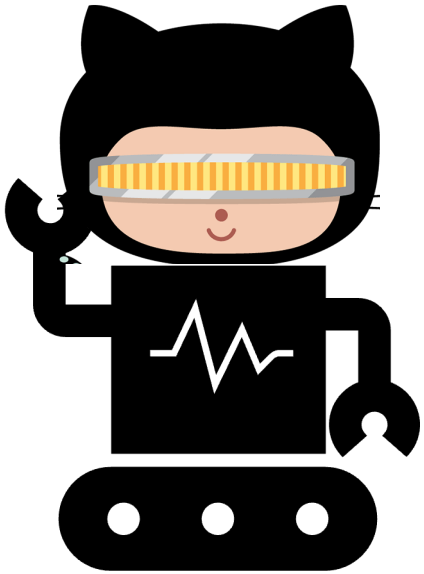


Agent-Based Automation of Collaborative Software Development

Current Status and Future Prospects

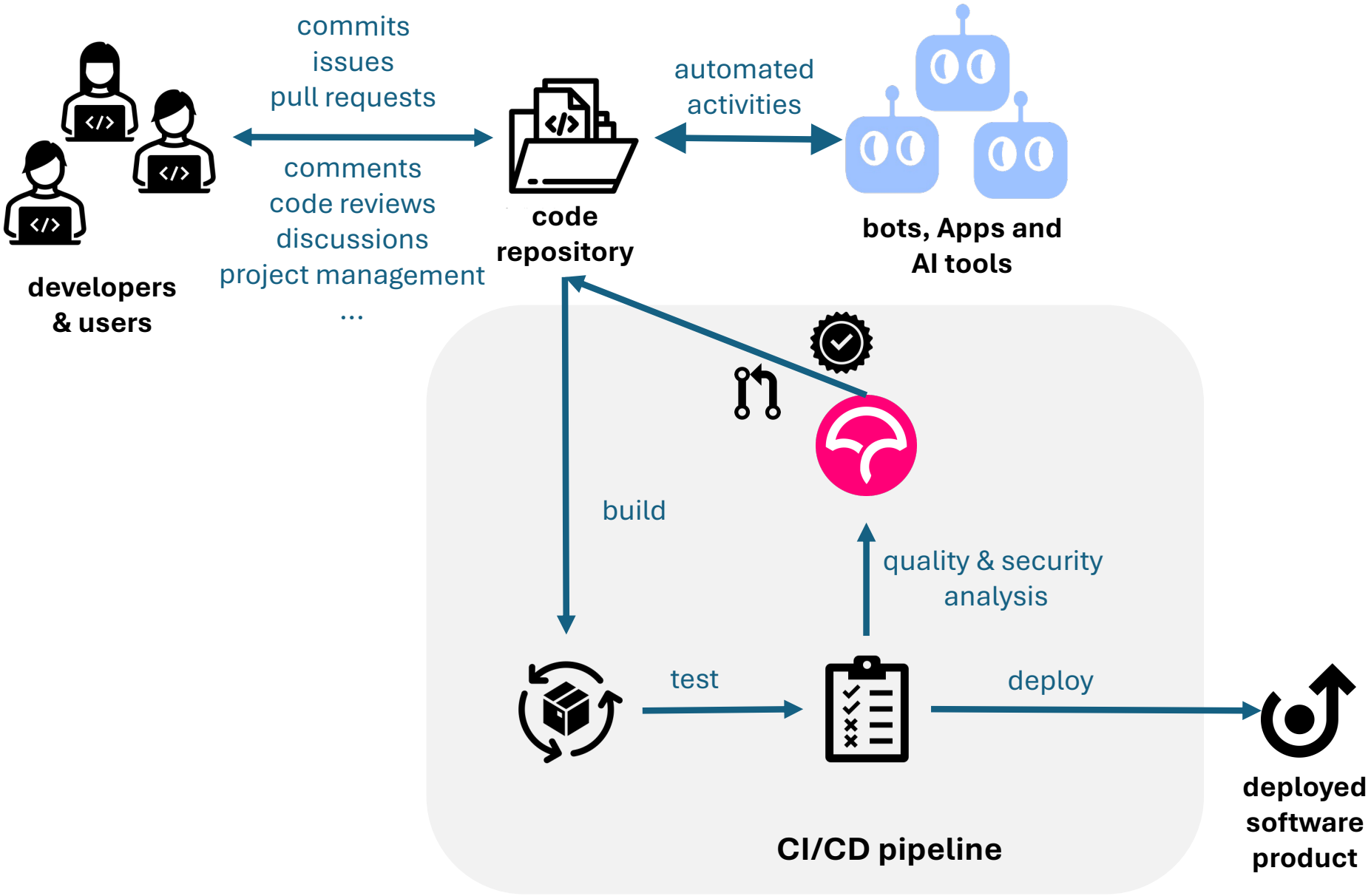
UMONS
Université de Mons

Tom Mens
tom.mens@umons.ac.be

Software Engineering Lab
Faculty of Sciences

Séminaire virtuel SOTELO: De l'impact de l'IA générative sur le génie logiciel
CEA, Saclay, France, le 4 juin 2025

Collaborative Software Development (CSD)



Today's Landscape of CSD Automation

How do geographically distributed teams automate their collaborative software development practices?

- Version control
- Cloud-based "social coding" platforms
- Package managers and package registries
- Social communication channels (e.g., Slack)
- Dependency, quality, performance and vulnerability analysers
- Automation tools (CI/CD, bots, AI assistants, ...)

Today's Landscape of CSD Automation

- Version control
  - Mostly Git, nowadays

- Cloud-based "social coding" platforms
  - Support issue tracking, pull requests, code reviewing, Git hosting, discussion fora, wiki, web pages, project management
  - Self-hosted or cloud-based

Today's Landscape of CSD Automation

- Dependency, quality, performance and vulnerability analysers
  - Dependency Analysis
    - Check and fix dependency issues: outdatedness, deprecation, vulnerabilities, unused or obsolete dependencies, ...
    - Tools: Dependabot (for GitHub), Renovate Bot (general purpose)
  - Code Quality Analysis
    - Use static or dynamic analysis to detect code smells, best practices, coding conventions, quality problems, potential or real bugs, test coverage, ...
    - Tools: CodeQL, SonarQube, Codacy, ...
  - Security and Vulnerability Scanning
    - Detect potential or real security weaknesses in code
    - Software composition analysis (SCA): Detect vulnerabilities in project dependencies
    - Tools: CodeQL (GitHub), Trivy, Snyk, OWASP Dependency-Check
  - Profiling
    - Detect and fix performance issues, memory leaks, ...

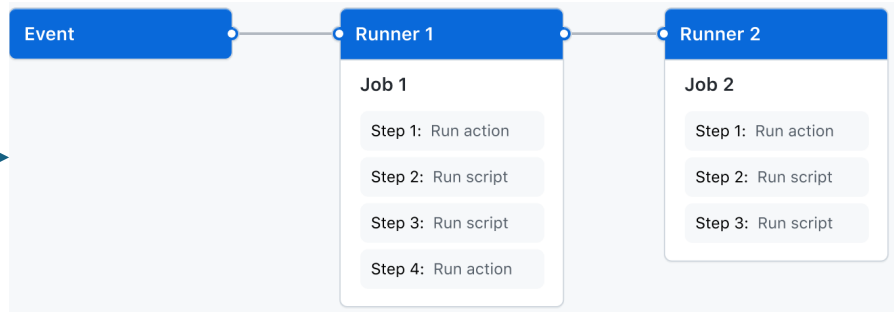
Today's Landscape of CSD Automation

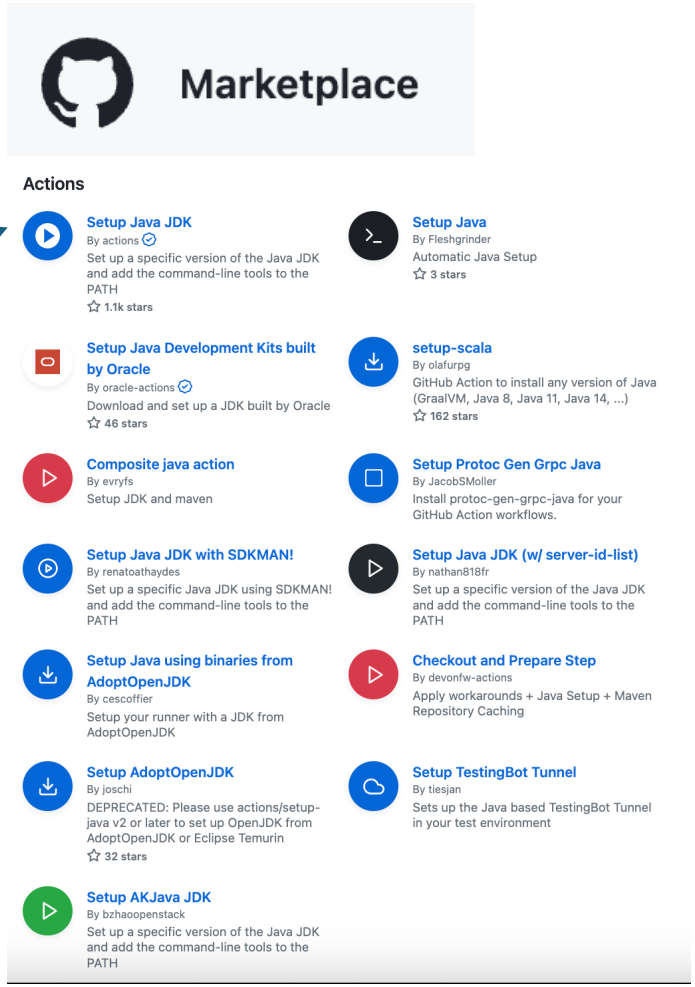
CI/CD Example: GitHub Actions workflow automation

Code repository
workflow config file

```yaml
name: Java CI with Maven
on:
  pull_request:
jobs:
  build:
    runs-on: ubuntu-latest
    steps:

    - name: Check out code
      uses: actions/checkout@v3

    - name: Set up JDK 17
      uses: actions/setup-java@v3

    - name: Build with Maven
      run: mvn -B package --file pom.xml
```

trigger on pull request



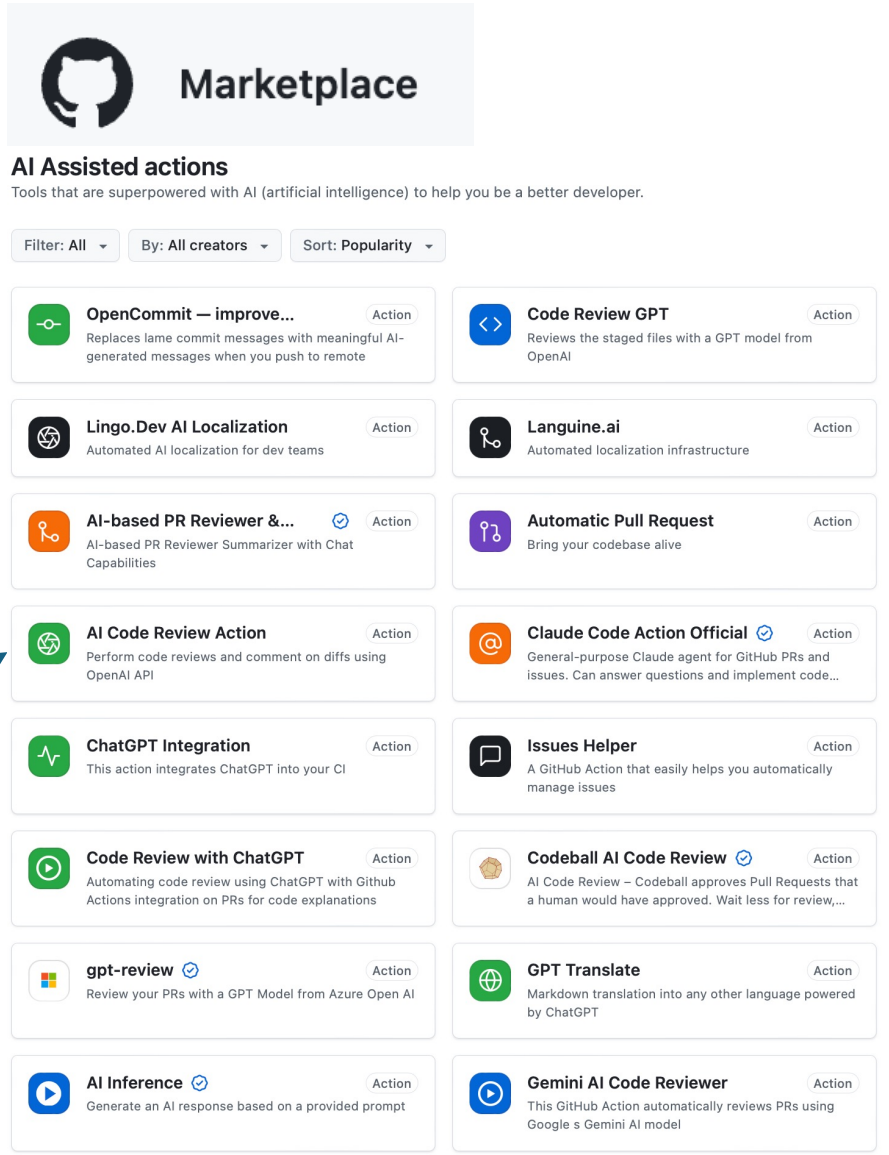


Today's Landscape of CSD Automation

CI/CD example: GitHub Actions workflow automation with AI assisted actions

Code repository
workflow config file

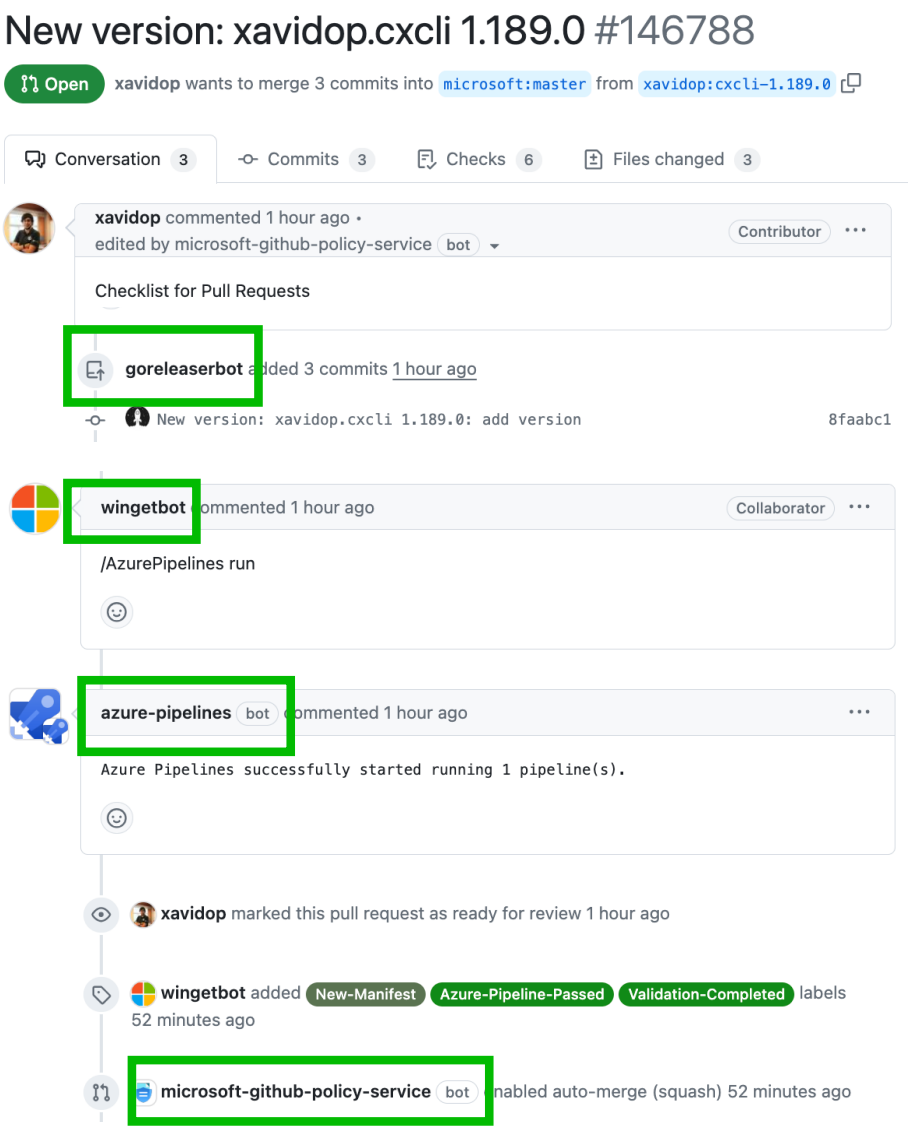
```yaml
name: AI Code Reviewer
on:
  pull_request:
    types:
      - opened
      - synchronize
permissions: write-all
jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout Repo
        uses: actions/checkout@v3

      - name: AI Code Reviewer
        uses: username/ai-code-reviewer@main
        with:
          GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
          OPENAI_API_KEY: ${{ secrets.OPENAI_API_KEY }}
          OPENAI_API_MODEL: "gpt-4"
```



Today's Landscape of Automation for Collaborative Software Development

Automation bots

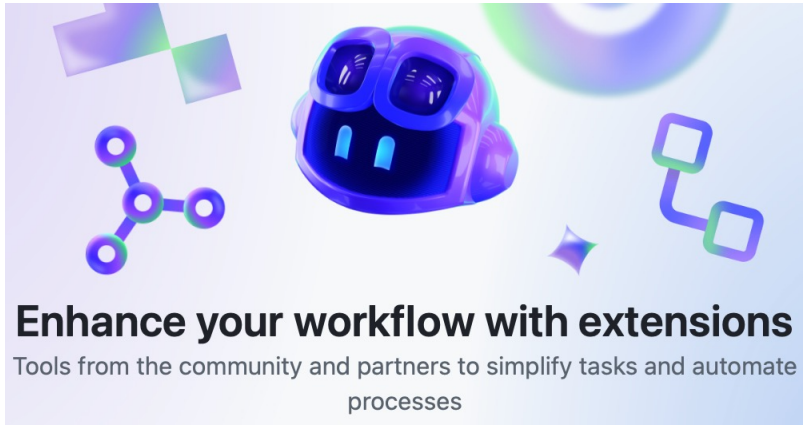
- Agents that interact with software contributors (through PR and issue comments) to support a wide range of activities.
- Today, these agents tend to focus on specific activities.

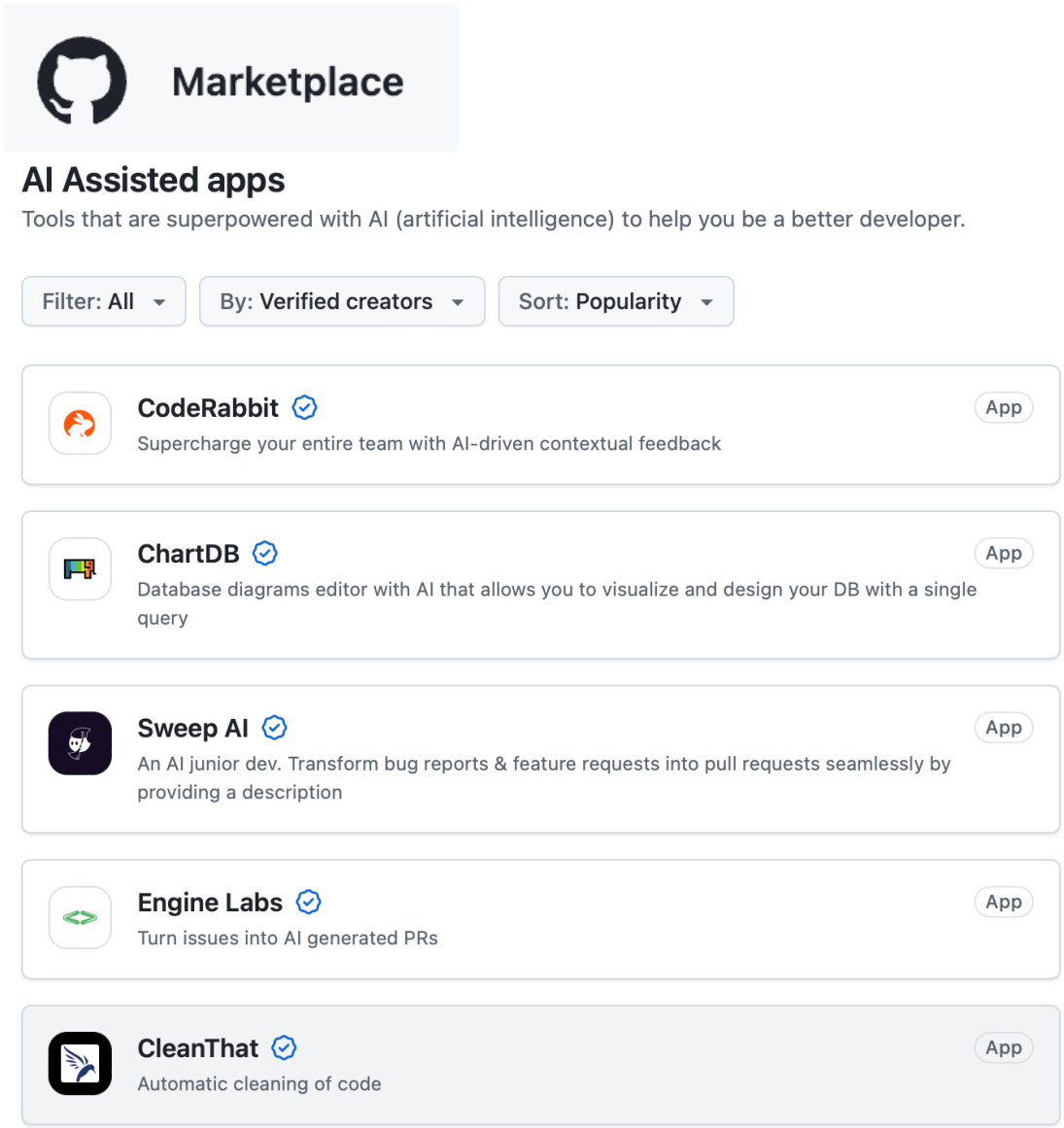


Today's Landscape of Automation for Collaborative Software Development

Automation bots

- Using third-party Apps
  (e.g. GitHub Marketplace)

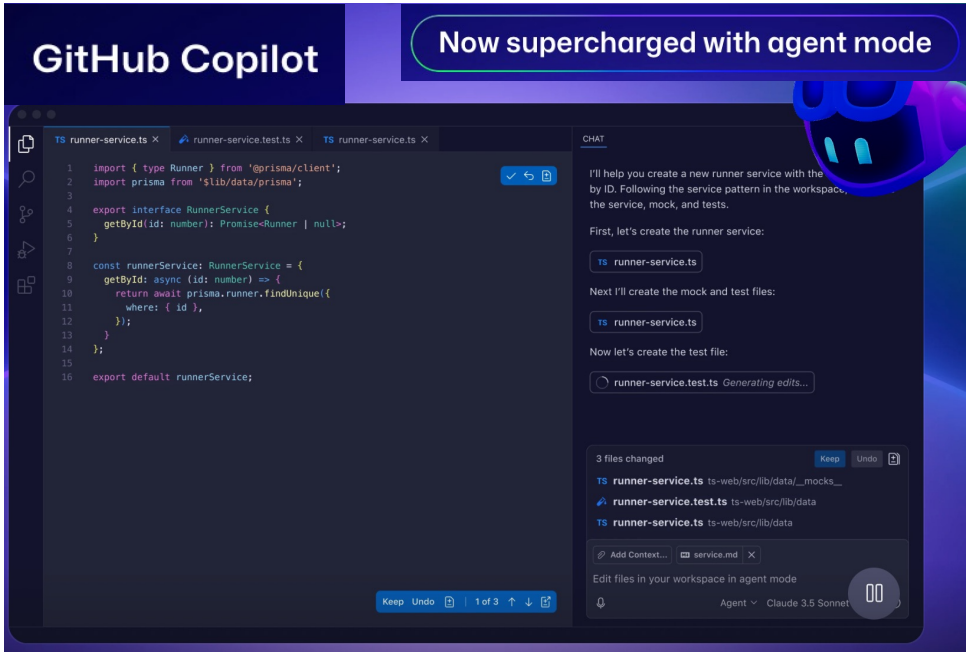


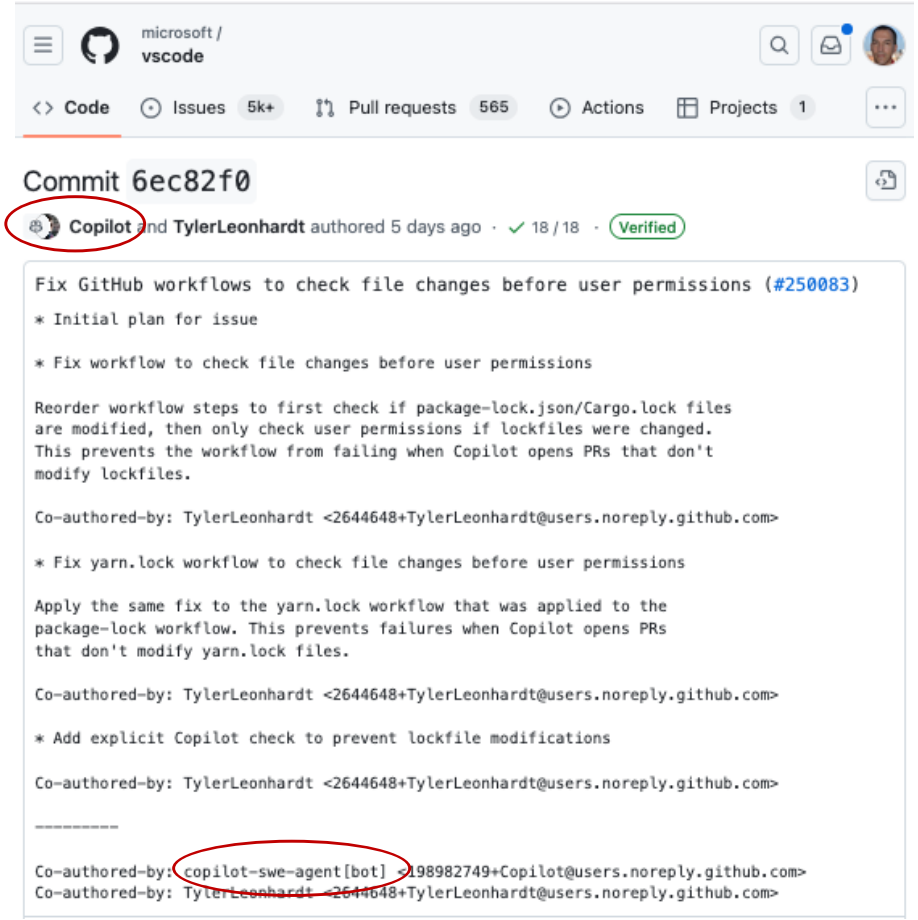


Today's Landscape of Automation for Collaborative Software Development

AI-powered coding assistants

- Generative AI to assist (collaborative) development tasks
- E.g. Github CoPilot, https://www.navto.ai/category/code-assistant





Today's Landscape of SE Automation

Where does AI fit in?

SE activity	SE task		Total
Requirements engineering	Anaphoric ambiguity treatment (4) Requirements analysis and evaluation (2) Coreference detection (1) Specification formalization (1) Use cases generation (1)	Requirements classification (4) Specification generation (2) Requirements elicitation (1) Traceability automation (1)	17
Software design	GUI retrieval (1) Software specification synthesis (1)	Rapid prototyping (1) System design (1)	4
Software development	Code generation (118) Code summarization (21) Code translation (12) API inference (5) API recommendation (5) Code representation (3) Method name generation (2) Agile story point estimation (1) API documentation smells (1) Data analysis (1) Control flow graph generation (1) Instruction generation (1) Others (14)	Code completion (22) Code search (12) Code understanding (8) Program synthesis (6) Code editing (5) Code comment generation (2) Code recommendation (2) API documentation augment (1) API entity and relation extraction (1) Fuzz driver generation (1) Identifier normalization (1) Type inference (1)	247
Software quality assurance	Vulnerability detection (18) Bug localization (5) Testing automation (4) Defect detection (2) Static analysis (2) Compiler fuzzing (1) Invariant prediction (1) Mobile app crash detection (1) Test prediction (1)	Test generation (17) Verification (5) Fault localization (3) GUI testing (2) Binary taint analysis (1) Decompilation (1) Malicious code localization (1) Resource leak detection (1)	66
Software maintenance	Program repair (35) Code review (7) Bug reproduction (3) Duplicate bug report detection (3) Log parsing (3) Sentiment analysis (3) API misuses repair (1) Bug triage (1) Code review explained (1) Crash bug repair (1) Incivility detection (1) Patch detection (1) Rename Refactoring (1) Technical debt payback (1) Web test repair (1) Others (5)	Code clone detection (8) Debugging (4) Review/commit/code classification (3) Logging (3) Code revision (2) Vulnerability repair (2) Bug prediction (1) Code coverage prediction (1) Code-Review defects repair (1) Dockerfile Repair (1) Patch correctness prediction (1) Program merge conflicts repair (1) Tag recommendation (1) Traceability recovery (1) Type error repair (1)	99
Software management	Effort estimation (2)	Software tool configuration (1)	3

Source: Hou et al. (2024)
Large Language Models for Software Engineering:
A Systematic Literature Review
ACM TOSEM 33(8), DOI:10.1145/3695988

Today's Landscape of CSD Automation
Where does AI fit in?

"Basic" software development support:

- Code completion (e.g., GitHub CoPilot)
- Quality and performance analysis and improvement
- Security analysis
- Documentation
- Q&A

"Collaborative" software development support:

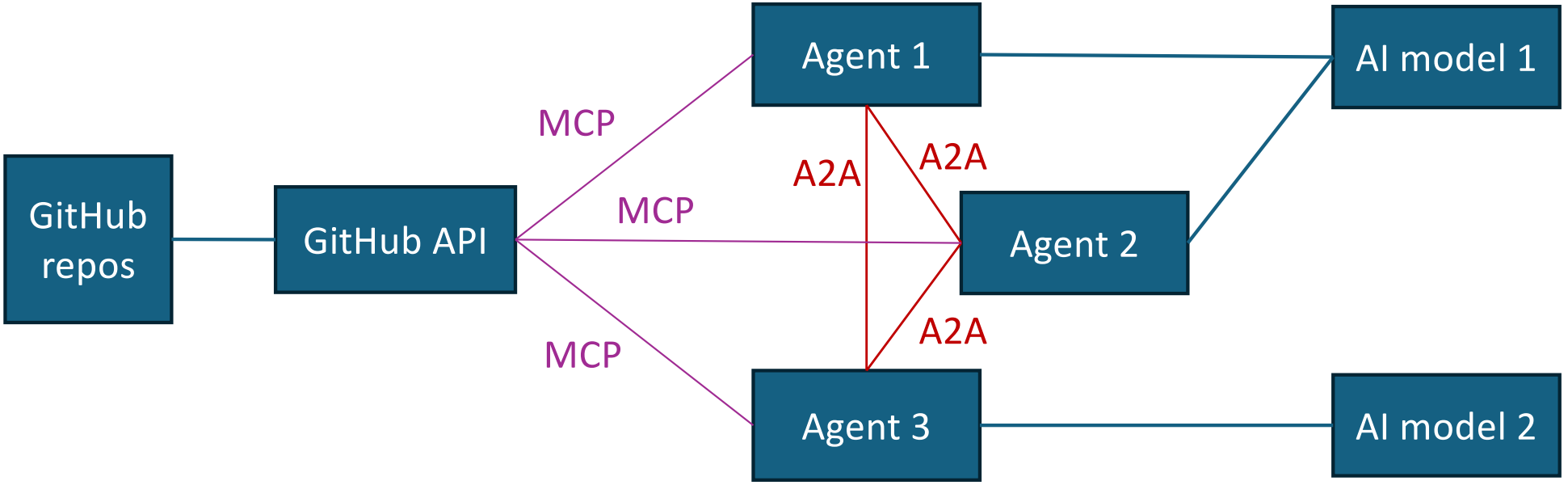
- Workflow optimisation
- Issue and pull request management
- Merge support
- Code review assistance

Tomorrow's Landscape of CSD Automation
How do we see AI being used in the future?

- Predictive/proactive AI
  - Anticipate and fix errors, bugs, vulnerabilities, incompatibilities, merge conflicts, ... before they occur
- Self-healing software
  - Fix identified software problems (bugs, vulnerabilities, dependency issues) fully autonomously
- Low-code development
  - Generate full code from a given specification, with minimal input
  - Modify existing code based on an issue report

Tomorrow's Landscape of CSD Automation
How do we see AI being used in the future?

- Multi-agent collaborative AI
  - AI agents will evolve from coding "assistants" to "autonomous" developers
  - AI agents will collaborate to carry out complex CSD tasks with little need of human intervention
  - Using open protocols to create smart interacting agents to carry out complex CSD tasks
    - **MCP** to connect AI models (e.g. LLM) to external tools such as GitHub API https://modelcontextprotocol.io
    - **A2A** to allow multiple autonomous agents to communicate https://google.github.io/A2A



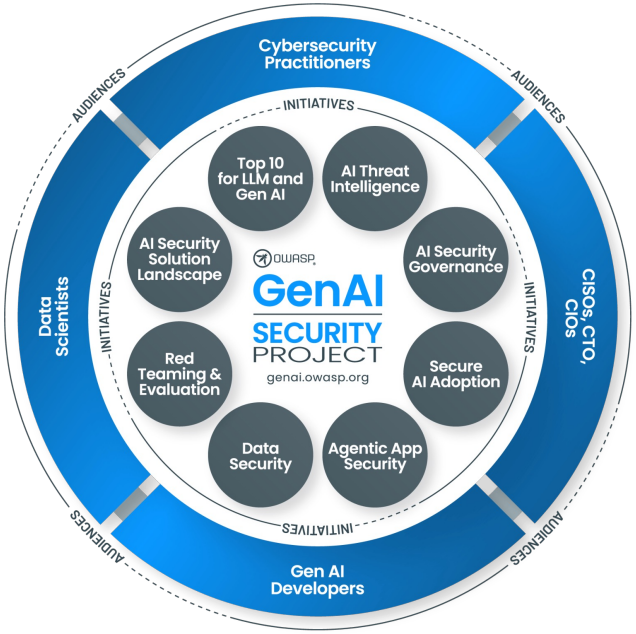
Threats of Using AI in CSD

Increased security risks

- AI-generated code may introduce non-obvious security flaws

https://owasp.org/www-project-top-10-for-large-language-model-applications/

https://genai.owasp.org



OWASP Top 10 for Large Language Model Applications version 1.1

LLM01: Prompt Injection

Manipulating LLMs via crafted inputs can lead to unauthorized access, data breaches, and compromised decision-making.

LLM02: Insecure Output Handling

Neglecting to validate LLM outputs may lead to downstream security exploits, including code execution that compromises systems and exposes data.

LLM03: Training Data Poisoning

Tampered training data can impair LLM models leading to responses that may compromise security, accuracy, or ethical behavior.

LLM04: Model Denial of Service

Overloading LLMs with resource-heavy operations can cause service disruptions and increased costs.

LLM05: Supply Chain Vulnerabilities

Depending upon compromised components, services or datasets undermine system integrity, causing data breaches and system failures.

LLM06: Sensitive Information Disclosure

Failure to protect against disclosure of sensitive information in LLM outputs can result in legal consequences or a loss of competitive advantage.

LLM07: Insecure Plugin Design

LLM plugins processing untrusted inputs and having insufficient access control risk severe exploits like remote code execution.

LLM08: Excessive Agency

Granting LLMs unchecked autonomy to take action can lead to unintended consequences, jeopardizing reliability, privacy, and trust.

LLM09: Overreliance

Failing to critically assess LLM outputs can lead to compromised decision making, security vulnerabilities, and legal liabilities.

LLM10: Model Theft

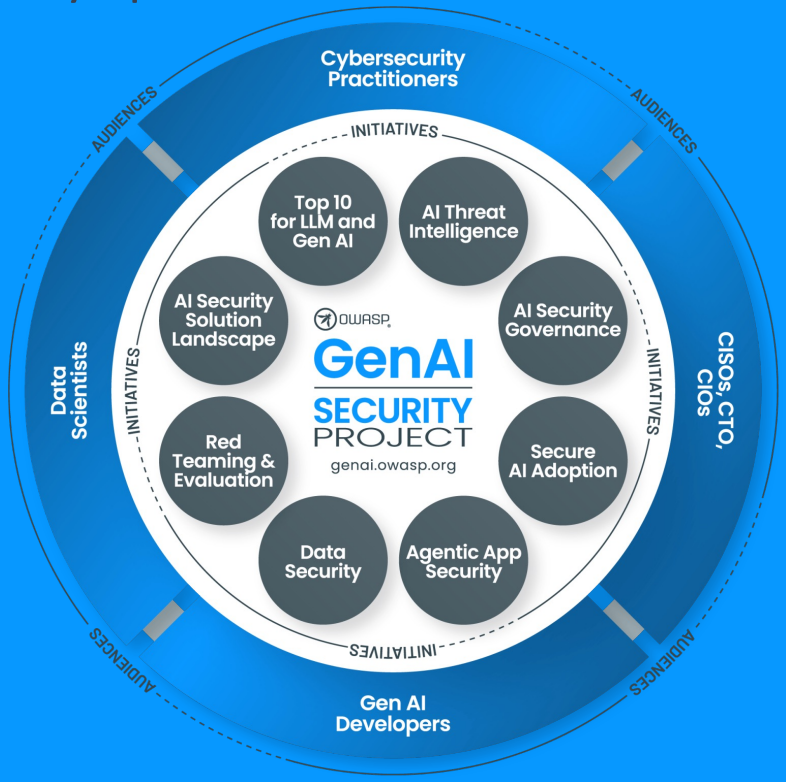
Unauthorized access to proprietary large language models risks theft, competitive advantage, and dissemination of sensitive information.

LLM03: 2025 Supply Chain

LLM supply chains are susceptible to various vulnerabilities, which can affect the integrity of training data, models, and deployment platforms. These risks can result in biased outputs, security breaches, or system failures. While traditional software vulnerabilities focus on issues like code flaws and dependencies, in ML the risks also extend to third-party pre-trained models and data.

Common Examples of Risks

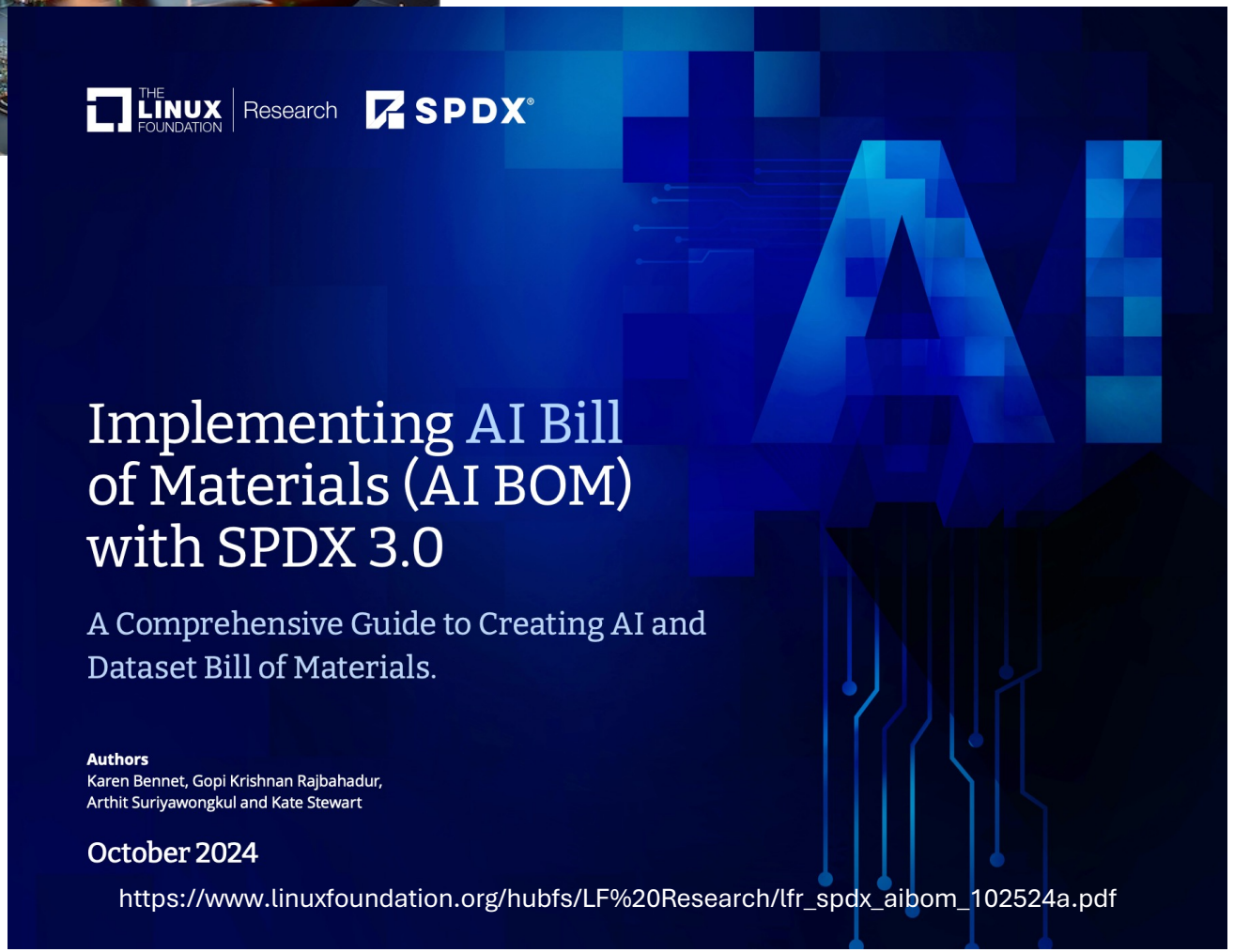
1. Traditional Third-party Package Vulnerabilities
2. Licensing Risks
3. Outdated or Deprecated Models
4. Vulnerable Pre-Trained Model
5. Weak Model Provenance
6. Vulnerable LoRA adapters
7. Exploit Collaborative Development Processes
8. LLM Model on Device supply-chain vulnerabilities
9. Unclear Terms and Conditions and Data Privacy Policies



https://genai.owasp.org

From SBOM to AIBom.

Integrating AI tools in the software supply chain adds more complexity and security risks, requiring to expand the security lens.



Threats of Using AI in CSD

- Suboptimal code
  - Complexity
    - AI-generated code may be too complex, inefficient, hard to maintain, undocumented, not following best practices
  - Incorrectness
    - AI models suffer from limitations and hallucinations and may generate code that looks valid but has fundamental, yet non-obvious errors.
  - Lack of transparency
    - Unverifiable solutions and difficult to debug if the underlying decision-making process is unknown

Threats of Using AI in CSD

- Lack of domain context
  - Each software project, process and product is different. AI suggestions may not always take into account the precise domain-specific context (“secret sauce”), leading to inappropriate suggestions.
- Loss of project knowledge
  - If AI is writing significant parts of the code base, core developers may no longer “know” or understand the code, making it more difficult to modify it.
- Loss of development skills
  - If AI is taking over coding, developers will loose critical thinking and programming experience, making them over-reliant on AI. This could lead to a vicious circle requiring them to use even more AI.
- Legal issues
  - Suggested code recommendations might include, or be based on, (unpermissively) licenced or copyrighted code.

Threats of Using AI in CSD

All more generic concerns of using AI apply as well:

- Trustworthiness
  - Overreliance: Putting **too much trust in AI** suggestions without thoroughly reviewing them.
  - Lack of trust: Developers might not trust perfectly valid AI-generated suggestions.
- Safety/Responsibility/Accountability
  - Who sould be held accountable if AI-generated code has problems?
- Learning bias
  - AI models are trained on existing data/code, and may not come up with innovative or groundbreaking solutions.
- Sustainability
  - AI consumes significant computational resources, raising environmental concerns.
- Model collapse
  - As more new code will become AI-generated, AI models will become trained on AI-generated data, leading to progressive degradation. This may amplify errors and quality problems in newly generated code resulting in less useful code.
- Ethical concerns
  - As training datasets may be biased, suggested code recommentations could be possibly discriminatory.

Reflections on the AI-pocalypse

Is there such a thing as too much automation?

- AI models introduce *yet another layer of*
  - *abstraction/complexity* in software development
  - bugs and security risks
- AI takes over entry-level tasks, *preventing new professionals from acquiring the necessary fundamentals, hands-on practice and experience* to master more complex tasks.
- Over-reliance on AI
  - perpetuates status quo thinking
  - slows down innovation (cf. David's presentation)
- What to do when automation fails (it will!)? Who will be able to understand that it fails, why it fails, and how to fix it?

Migitating AI Threats in CSD

- Human audits
  - Humans should review all AI-generated code for quality, bias, security, inconsistencies, bugs, instead of blindly accepting its suggestions.

  But will they? (Humans may be lazy, under time pressure, underpaid, ...)

  And will the gain in auto-generated code compensate for this extra reviewing effort? One is taking the "fun" part out of coding to replace it by the "boring" part of reviewing...

- Awareness and training
  - Developers (and managers!) need to be trained on the limitations of AI
  - Humans should be trained in keeping their critical thinking
  - Juniors should be trained in fundamental technical skills even if AI tools exist that already automate them

  But will they?

- Good training data
  - AI tools needs to be trained on high-quality, non-malicious, unbiased datasets to avoid the garbabe-in-garbage-out phenomenon

  But is it feasible to create such "big enough" datasets?

Migitating AI Threats in CSD

- Better traceability
  - Impose AI BOMs with standardised formats and tools
- Better regulations
  - Regulatory standards need to be proposed for responsible and ehtical AI usage in software development.
  - AI providers and AI users need to be enforced to comply to these standards.

  But how to check that they do? And what to do if they don't?
- Improved AI tooling
  - Explainable AI: AI generated code should come with explanations to understand how and why suggestions were made
  - AI tools need to be made more environmentally friendly, less biased, more context-aware, more responsible, less hallucinatory, ...

  But this is not something that users of the AI tools have much control over...