КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 4 (32), 2026
ISSN 2663 – 4023
DOI 10.28925/2663-4023.2026.32.1187
УДК 004.056.5

Вахула Олександр Петрович
аспірант, асистент кафедри «Захист інформації»
Національний університет «Львівська Політехніка», Львів, Україна
ORCID: 0009-0008-5367-3344
oleksandr.p.vakhula@lpnu.ua

Марчук Дмитро Андрійович
студент кафедри захисту інформації
Національний Університет «Львівська Політехніка», Львів, Україна
ORCID: 0009-0007-7385-8752
dmytro.marchuk.kb.2024@gmail.com

ПІДХІД SECURITY-AS-CODE ДЛЯ АВТОМАТИЗАЦІЇ ВІДПОВІДНОСТІ PCI DSS З ВИКОРИСТАННЯМ АВТОНОМНИХ АГЕНТІВ НА БАЗІ ВЕЛИКИХ МОВНИХ МОДЕЛЕЙ

Анотація. Статтю присвячено дослідженню підходу Security-as-Code для автоматизації забезпечення відповідності стандарту PCI DSS у хмарно-нативних середовищах із використанням автономних агентів на базі великих мовних моделей. На основі аналізу PCI DSS v4.0.1, який визначає 12 принципових вимог і орієнтований на безперервні оцінювання, доказове підтвердження контролів і коректне визначення області застосування, запропоновано референтну архітектуру, що інтегрує декларативні політики безпеки, механізми превентивного контролю конфігурацій і runtime-моніторинг поведінки систем. У роботі формалізовано механізм трансформації нормативних вимог у машинно-перевірювані політики через використання контрольних специфікацій, структурованої бази знань і підходу Retrieval-Augmented Generation, що забезпечує трасованість походження політик і зменшення ризику некоректної генерації. Запропоновано модель автономних агентів, які виконують функції трансформації вимог у політики, аналізу відхилень конфігурацій, збору й нормалізації доказів, а також формування рекомендацій щодо усунення порушень із використанням контрольованих механізмів внесення змін. Визначено метрики оцінювання безперервної відповідності, включаючи покриття контролів, латентність виявлення відхилень, показники MTTD та MTTR, повноту доказової бази та точність детекції. Запропоновано план експериментальної валідації архітектури в тестовому середовищі, що моделює Cardholder Data Environment. Запропонований підхід дозволяє забезпечити безперервну, перевірювану та масштабовану відповідність PCI DSS у хмарно-нативних середовищах.

Ключові слова: PCI DSS; Security-as-Code; Policy-as-Code; Compliance-as-Code; автономні агенти; великі мовні моделі; DevSecOps; Gatekeeper; безперервна відповідність.

ВСТУП

Стандарт PCI DSS визначає сукупність технічних і організаційних вимог для захисту даних платіжних карток у середовищах, де такі дані зберігаються, обробляються або передаються. Метою стандарту є забезпечення конфіденційності та цілісності платіжних даних шляхом впровадження контролів безпеки, які охоплюють конфігурацію систем, управління доступом, моніторинг подій, управління вразливостями та інші аспекти захисту інформаційних систем. Особливістю сучасної версії стандарту є орієнтація не лише на досягнення відповідності, а й на її підтримання протягом усього життєвого циклу системи.

Практична реалізація безперервної відповідності ускладнюється використанням хмарно-нативних технологій, які характеризуються високою динамічністю. Інфраструктура визначається як код, конфігурації змінюються автоматизовано через конвеєри CI/CD, а обчислювальні ресурси створюються та знищуються динамічно. У таких умовах періодичні аудити не забезпечують повного відображення фактичного стану системи між моментами перевірок. Крім того, підготовка доказової бази для підтвердження виконання контролів часто потребує значних ручних зусиль і створює додаткове навантаження на команди розробки та безпеки.

Зростання складності сучасних інформаційних систем і вимог до їх безпеки стимулює розвиток підходів, спрямованих на автоматизацію процесів забезпечення відповідності. Одним із таких підходів є використання концепції Everything-as-Code, яка передбачає представлення інфраструктури, конфігурацій і політик безпеки у формі машинно-оброблюваного коду. У цьому контексті підходи Policy-as-Code, Security-as-Code та Compliance-as-Code дозволяють формалізувати вимоги безпеки та відповідності у вигляді декларативних політик, які можуть бути автоматично застосовані, перевірені та відтворені.

Паралельно з розвитком підходів до кодифікації політик безпеки відбувається інтеграція систем штучного інтелекту, зокрема великих мовних моделей, у процеси управління безпекою. Такі системи можуть виконувати функції аналізу вимог, генерації політик, збору доказів і виявлення відхилень. Використання автономних агентів на базі великих мовних моделей відкриває можливість автоматизації складних багатостадійних процесів забезпечення відповідності, включаючи трансформацію нормативних вимог у формалізовані політики, моніторинг стану системи та підготовку аудиторської документації.

Водночас використання автономних агентів у домені безпеки потребує спеціальних механізмів контролю, які забезпечують передбачуваність і перевірюваність їх дій. Зокрема, рішення агентів мають базуватися на детермінованих політиках і виконуватися через контрольовані інтерфейси, що дозволяє зменшити ризики некоректних або небезпечних дій. Забезпечення прозорості, трасованості та доказовості виконання контролів є критично важливим для підтвердження відповідності стандартам безпеки.

У цьому контексті актуальним є дослідження архітектурних підходів, які поєднують декларативні політики безпеки, автоматизований моніторинг, доказове забезпечення та інтелектуальні механізми автоматизації на основі автономних агентів. Такий підхід дозволяє перейти від періодичного до безперервного забезпечення відповідності, підвищити прозорість і відтворюваність процесів контролю та зменшити залежність від ручних процедур.

Огляд літератури. Стандарт PCI DSS v4.0.1 визначає відповідність як сукупність технічних і операційних контролів, наголошуючи на необхідності коректного визначення області застосування (scoping), мережевої/системної сегментації та зменшення «поверхні» обробки й зберігання даних облікових записів (мінімізація точок, де можуть з’являтися account data) [1, 8]. У цьому контексті scoping виступає не формальною процедурою, а інженерною дисципліною, що впливає на масштаб контрольних заходів, їхню перевірюваність та обсяг доказів, які потрібно формувати для аудитних процедур.

Підходи до практичного виконання PCI DSS у хмарних середовищах доповнюються настановами PCI SSC, які деталізують питання розподілу ролей і відповідальності між споживачем та провайдером, уточнюють особливості визначення

області застосування (з урахуванням моделей спільної відповідальності), описують типові виклики досягнення відповідності в хмарі та підкреслюють значущість процесів реагування на інциденти. [6, 14]. Сукупно ці матеріали формують методичну основу для «декомпозиції» PCI DSS на контрольні артефакти: частина контролів може бути формалізована й автоматизована (оптимально – у вигляді коду та перевірок у CI/CD), тоді як організаційні процедури можуть підтримуватися автоматизованим збором доказів, журналюванням і керованим workflow погоджень/винятків.

У домені Security-as-Code (SaC) описано підходи, що адаптують стандарти управління безпекою до DevSecOps-практик шляхом кодифікації контролів і перенесення їхньої перевірки в автоматизовані пайплайни. Зокрема, у роботах з SaC показано, що формалізація безпечних конфігурацій як коду та їхня систематична перевірка (policy tests, контроль дрейфу, автоматизоване підтвердження стану) підвищують повторюваність і масштабованість впровадження вимог, зменшуючи залежність від ручних аудитних чек-листів [15, 16]. Окремо розглядається застосовність SaC у cloud-native середовищах на основі Kubernetes, де динамічність змін (короткі цикли релізів, декларативні маніфести, горизонтальне масштабування) робить автоматизацію контролів практично необхідною умовою підтримки безперервної відповідності [16, 17].

Policy-as-Code (PaC) розглядається як підхід до формалізації правил доступу та управління (зокрема RBAC/ABAC) у вигляді коду з інтеграцією цих правил у пайплайни й контури виконання, де підкреслюються масштабованість, узгодженість, безперервний контроль і міжхмарна інтероперабельність [17, 18]. Такий підхід напряму узгоджується з вимогами PCI DSS щодо контролю доступу (включно з принципом «need to know») та ідентифікації/автентифікації, оскільки формалізовані політики можуть оцінюватися детерміновано і відтворювано, що підвищує аудитопридатність порівняно з неформалізованими організаційними політиками.

У прикладній площині OPA позиціонується як рушій оцінювання політик із декларативною мовою Rego, придатною для опису правил над структурованими даними (API-запити, IaC, конфігурації) [2, 9]. Інструмент Gatekeeper як спеціалізована інтеграція OPA з Kubernetes описується як validating/mutating webhook, який застосовує політики у вигляді CRD-орієнтованих обмежень (constraints) і відокремлює прийняття policy-рішень від внутрішньої логіки API-сервера Kubernetes, що є принциповим для масштабованого governance [3, 10, 11]. У результаті зв’язка OPA/Gatekeeper найбільш природно підтримує превентивний контроль конфігурацій та змін, які проходять через Kubernetes API (тобто «до» фактичного виконання навантаження).

Попри ефективність admission-контролю, PCI DSS містить значний блок вимог до журналювання, моніторингу та тестування безпеки, що потребує контролів «під час виконання» (runtime) [1, 18]. У Kubernetes значуща частина критичних подій відбувається після розгортання (наприклад, exec у контейнер, аномальні системні виклики, підозріла поведінка процесів), і ці події не завжди повністю охоплюються превентивними політиками admission-рівня. У цьому контексті Falco описується як хмарно-нативний інструмент runtime security для виявлення аномальної поведінки в реальному часі у хостах, контейнерах та Kubernetes [4, 12]. Для маршрутизації та інтеграції подій Falco використовується Falcosidekick як проксі-forwarder для HTTP output та надсилання алертів до широкого спектра інтеграцій [13, 19]. Окремі дослідження аналізують кілька моделей взаємодії Falco та OPA в Kubernetes, включно з прямою передачею подій через Falcosidekick до OPA API, застосуванням OPA як admission controller для блокування небезпечних конфігурацій, а також використанням

проміжного сервісу для складної обробки подій і виконання реакцій (ізоляція/видалення ресурсів тощо) із принципом мінімальних привілеїв у розподілі ролей компонентів [14]. Така композиція є концептуально узгодженою з потребами PCI DSS: превентивні контролі доповнюються безперервним моніторингом і керованим реагуванням, що залишає аудитний слід у журналах та формує доказову базу.

У площині керування комплаєнс-процесами в хмарі запропоновано концепції автоматизованої перевірки відповідності (conformity verification), що розглядають відповідність як формалізований процес зі структурованими перевірками й артефактами доказів [19, 20]. Також досліджуються ризики Shadow IT у публічній хмарі та загрози, пов’язані з неконтрольованим зберіганням секретів у вихідному коді, що безпосередньо корелює з вимогами PCI DSS щодо управління доступом, захисту облікових даних і безпечної розробки [20, 21, 22]. Додатково аналізується підхід SOAR до автоматизації управління інцидентами у хмарі, який є близьким до «agentic response» як процесу, але вимагає строгих політик, оркестрації та контрольованих повноважень виконавчих компонентів [21, 22].

Паралельно з еволюцією SaC/PaC, інтеграція генеративного ШІ у життєвий цикл розробки програмного забезпечення формує новий клас ризиків (галюцинації, supply chain-атаки через пакети, витоки даних), що потребує переосмислення ролей людини та автоматизованих агентів у SDLC і валідаційних практик. У дослідженнях LLM-агентних систем показано, що поєднання reasoning і acting (ReAct) може зменшувати галюцинації завдяки взаємодії з зовнішніми джерелами знань, а RAG-архітектури додають «непараметричну пам’ять» для актуалізації знань та підвищення доказовості відповідей [23, 28, 29]. Водночас OWASP і NIST акцентують на необхідності керування ризиками ШІ та захисту LLM/agent-додатків від prompt injection та інших класів загроз, що є критичним для сценаріїв автономного виконання дій у середовищах з вимогами високої довіри, таких як PCI-контури [5, 25, 26, 27].

Проблематика, мета, об'єкт і предмет дослідження. Підтримка безперервної відповідності PCI DSS у хмарно-нативних середовищах (контейнеризація, Kubernetes, IaC, швидкі CI/CD-цикли) ускладнюється динамічністю інфраструктури та частими змінами конфігурацій. PCI DSS v4.0.1 прямо орієнтує організації на коректне визначення області застосування (scoping), сегментацію й мінімізацію місць появи account data, що визначає обсяг контролів, тестових процедур і доказів (evidence) [1]. Додаткові настанови PCI SSC для хмарних середовищ акцентують розподіл ролей і відповідальності, специфічні ризики відповідності в хмарі та вимоги до інцидент-реагування [6]. У результаті виникає практичний розрив між нормативними вимогами (текстові формулювання, процедурні очікування) та їхнім технічним виконанням (політики, конфігурації, моніторинг, журналювання), а також між фактичним станом системи і доказовою базою для аудиту (evidence completeness і traceability) [1].

Парадигми Security-as-Code і Policy-as-Code пропонують «кодифікувати» значну частину контролів як версіоновані й тестовані артефакти, інтегровані у DevSecOps-пайплайни, підвищуючи повторюваність і масштабованість контролів [16, 17]. Практичну основу Policy-as-Code складають механізми детермінованої оцінки правил у вигляді коду (наприклад, entity ["organization","Open Policy Agent","policy engine"] та мова Rego) і превентивне застосування політик у entity ["organization","Kubernetes","container orchestration"] через admission-контроль (наприклад, entity ["organization","Gatekeeper","opa kubernetes admission"]) [9, 10, 11]. Водночас значна частина вимог PCI DSS охоплює журналювання та безперервний моніторинг, що вимагає runtime-рівня спостереження: зокрема, entity

["organization","Falco","runtime security tool"] для детекції аномальної поведінки і entity ["organization","Falcosidekick","falco event forwarder"] для маршрутизації подій [1, 12, 13]. Додатковий виклик пов'язаний із впровадженням автономних агентів на базі LLM: хоча RAG/agentic-підходи можуть прискорити трансформацію вимог у політики та автоматизувати збір доказів [28, 29], вони створюють специфічні ризики (галюцинації, prompt injection, небезпечне виконання інструментів), і тому потребують окремих механізмів гарантування детермінованості рішень, трасованості дій і захисту контурів виконання [25, 26, 27]. У статті необхідно явно пов'язати ці LLM-ризики з вимогами PCI DSS щодо scoping, evidence та logging як критичними «точками контролю» [1, 25].

Мета дослідження – розробити та обґрунтувати архітектуру Security-as-Code для автоматизації відповідності PCI DSS у cloud-native середовищах із використанням автономних LLM-агентів за принципом *policy-first*, забезпечивши: формальну трансформацію вимог PCI DSS у Policy-as-Code; (безперервний контроль змін (CI/CD + admission) і runtime-моніторинг; автоматизований збір, нормалізацію та трасованість доказів для аудитних процедур; (iv) керування ризиками LLM (галюцинації, prompt injection) через RAG, безпечну оркестрацію інструментів і детерміновані механізми прийняття рішень [1, 9, 12, 25, 28].

Об'єктом дослідження є процес забезпечення та підтримки відповідності PCI DSS у хмарно-нативній інфраструктурі (Kubernetes/IaC/CI/CD), включно з контролями конфігурацій, моніторингом, журналюванням і формуванням доказової бази [1, 6].

Предметом дослідження є методи, моделі та засоби реалізації Security-as-Code/Policy-as-Code і агентної автоматизації (LLM-агенти з RAG/інструментами) для: (a) формалізації PCI DSS-вимог у політики; (b) превентивного застосування політик; (c) класифікації runtime-подій; (d) автоматизації evidence-циклу; (e) мінімізації ризиків LLM у середовищах підвищеної довіри [9, 10, 12, 25, 28].

Теоретично й практично важливо розрізняти SaC та PaC, оскільки в автоматизації PCI DSS вони відіграють різні ролі: SaC часто охоплює повний життєвий цикл контролів (включно з перевірками, моніторингом, реагуванням і доказами), тоді як PaC є спеціалізованою підмножиною – формалізацією політик/правил у коді для детермінованого оцінювання. Таке розрізнення узгоджується як з інженерними підходами OPA/Rego (PaC), так і з більш широкими SaC-рамками, що розглядають автоматизацію безпеки через DevSecOps і CI/CD [9, 16, 23, 24].

Таблиця 1

Порівняння підходів Security-as-Code та Policy-as-Code у задачах відповідності PCI DSS

Критерій	Security-as-Code	Policy-as-Code
Об'єкт кодифікації	Комплекс контролів: політики, перевірки конфігурацій, сканування, моніторинг, реагування, збирання доказів, звітність	Декларативні правила/обмеження для прийняття policy-рішень (allow/deny, класифікація, відповідність)
Основна мета	Безперервний контроль безпеки й відповідності, зменшення «security debt», підготовка аудиту через автоматизацію доказів	Детерміноване, повторюване оцінювання політик над структурованими даними
Типові enforcement points	CI/CD gates, IaC-сканери, admission-контроль, runtime-детекція, SIEM/SOAR	Admission controller/webhook, внутрішній PDP у мікросервісах, IaC policy checks
Артефакти	Контрольні пакети, політики, тести, сценарії реагування, playbooks, evidence schemas	Rego-політики, Gatekeeper constraints, policy bundles
Сильні сторони	«End-to-end» автоматизація: від вимоги → до політики → до доказу → до звіту	Висока формалізація й аудитопридатність рішень; простота

Критерій	Security-as-Code	Policy-as-Code
		інтеграції як бібліотеки/сервісу
Обмеження	Вимагає ширшої екосистеми інструментів; складніший дизайн керування ризиками та доступами	Обмежене охоплення «runtime реальності» без додаткових сенсорів/моніторингу
Найкраще застосування для PCI DSS	Безперервна відповідність: контроль конфігурацій + моніторинг + доказова база + реагування	Превентивні правила (наприклад, TLS, привілеї контейнерів) та формалізація рішень за подіями

РЕЗУЛЬТАТИ ДОСЛІДЖЕННЯ

Методологія дослідження складається з трьох взаємопов'язаних блоків: (1) аналіз нормативної бази PCI DSS v4.0.1 та супутніх guidance документів (scoping, cloud guidelines); (2) систематизація інструментальних підходів PaC/SaC у cloud-native (OPA/Gatekeeper, Falco/Falcosidekick) та узагальнення практичних архітектур інтеграції; (3) дизайн архітектури «LLM agents + deterministic policy engines» з формалізацією артефактів, інтерфейсів і планом експериментальної валідації [1, 6, 9, 10, 12, 15, 25].

Ключовим принципом є policy-first: автономні агенти можуть генерувати, підказувати, планувати, узгоджувати та збирати докази, але рішення про відповідність/невідповідність і дії «enforcement» мають бути детермінованими й трасованими. Це особливо важливо для аудиту PCI DSS, де доказова база має бути відтворювана, перевірювана та пов'язана з конкретними контролями [1, 9, 26].

Концепція контрольної специфікації ControlSpec. Для формальної трансформації вимог PCI DSS у машинно-перевірювані політики пропонується проміжна сутність ControlSpec (контрольна специфікація), яка поєднує «нормативний рівень» і «технічний рівень» у форматі YAML/JSON і зберігається в репозиторії разом із політиками та тестами. ControlSpec є відповіддю на розрив між «текстовою» вимогою і «кодом» політики та забезпечує трасованість.

Концептуально це узгоджується з підходами SaC, які пропонують кодифікацію контролів для повторюваного впровадження та перевірки [16, 17, 27].

Мінімальний набір полів ControlSpec:

- control_id: наприклад, PCI-4 або деталізовано PCI-4.x (внутрішній ідентифікатор мапінгу);
- pci_requirement_ref: посилання на принципову вимогу/підвимогу (для трасованості у звіті);
- control_intent: стислий опис цілі контролю;
- scope_selector: опис області застосування (namespace/labels/CDE-segment);
- policy_targets: перелік «точок контролю» (CI/IaC, admission, runtime, cloud config);
- policy_templates: посилання на конкретні політики (Rego, Gatekeeper constraints), параметри;
- evidence: опис даних, що збираються (логи, конфігурації, результати сканів);
- tests: тести політик (unit/integration), сценарії негативних/позитивних кейсів;
- exceptions: винятки з обґрунтуванням і термінами перегляду.

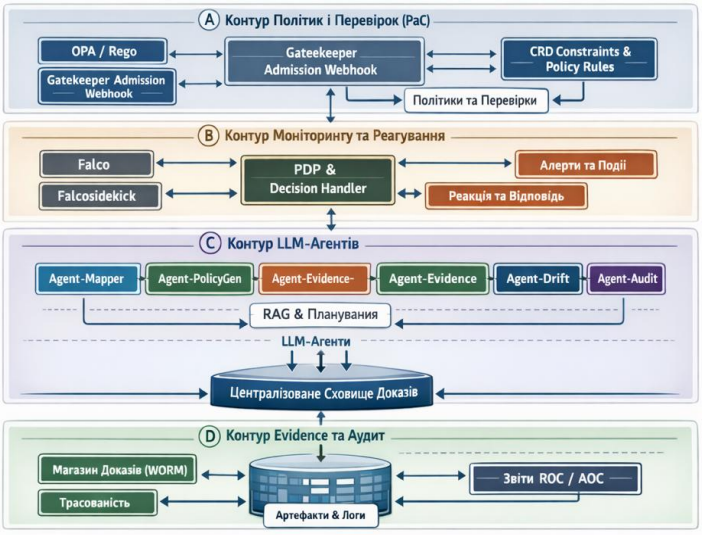

Рис. 1 Багатошарова архітектура системи Security-as-Code з автономними LLM-агентами, виконанням політик та збором аудиторських доказів

Запропонована архітектура є багатошаровою і складається з чотирьох контурів: (A) контур політик і перевірок (PaC); (B) контур моніторингу та реагування (runtime + response); (C) контур автономних LLM-агентів (інтелектуальний шар, RAG, планування); (D) контур evidence та аудитної звітності (evidence store, трасованість, артефакти ROC/AOC). (A) Policy-as-Code контур базується на OPA/Rego як механізмі декларативних політик для структурованих даних [2, 9]. Для Kubernetes інтеграція здійснюється через Gatekeeper – admission webhook, який застосовує політики як CRD-based constraints [3, 10, 11]. (B) Runtime та response контур використовує Falco для детекції небезпечної поведінки у реальному часі, а для пересилання подій – Falcosidekick як проксі-forwarder [12, 13, 28]. Події оцінюються політиками (OPA як PDP), а виконання реакції відбувається у відокремленому сервісі реагування (decision handler) з мінімально необхідними привілеями, що прямо відповідає концепції розподілу ролей та мінімізації привілеїв, яку описують Дарієнко й Когут у контексті Falco↔OPA інтеграції. (C) Контур автономних LLM-агентів реалізує спеціалізовані ролі:

- Agent-Mapper (PCI→ControlSpec): перетворює текст вимоги на структурований ControlSpec з параметрами області застосування та типами перевірок.
- Agent-PolicyGen (ControlSpec→Rego/Constraints): генерує або модифікує політики за шаблонами, додає тести, запускає «policy unit tests».
- Agent-Evidence (Evidence Collector): збирає докази з визначених джерел, нормалізує у схему evidence JSON, підписує метадані.
- Agent-Drift (Continuous Monitoring): аналізує поточний стан конфігурацій/активів та порівнює з еталонами, створює задачі/інциденти.
- Agent-Remediate (Safe PR Agent): готує pull request з виправленням IaC/маніфестів, але не зливає без перевірок і схвалення.
- Agent-Audit (Report Builder): компілює контрольний пакет (evidence + mapping + результати тестів) у структуру для ROC/AOC-шаблонів.

Наукова база для такого підходу включає RAG як механізм підкріплення генерації зовнішньою пам’яттю (для зменшення галюцинацій і забезпечення «provenance» знань)

та ReAct як шаблон поєднання reasoning і tool-based дій [23, 28, 29]. Водночас для безпечної експлуатації агентів необхідні керовані «інструменти» й захист від prompt injection як одного з ключових ризиків для LLM/агентних систем [25, 26, 29].

(D) Evidence та аудит передбачає централізований репозиторій доказів з незмінністю (WORM/append-only), де кожен доказ прив'язаний до control_id, часу, середовища та джерела. Такий підхід є необхідним, оскільки PCI DSS процедури оцінювання покладаються на перевірювані свідчення виконання контролів у часі й за областю застосування [1, 8].

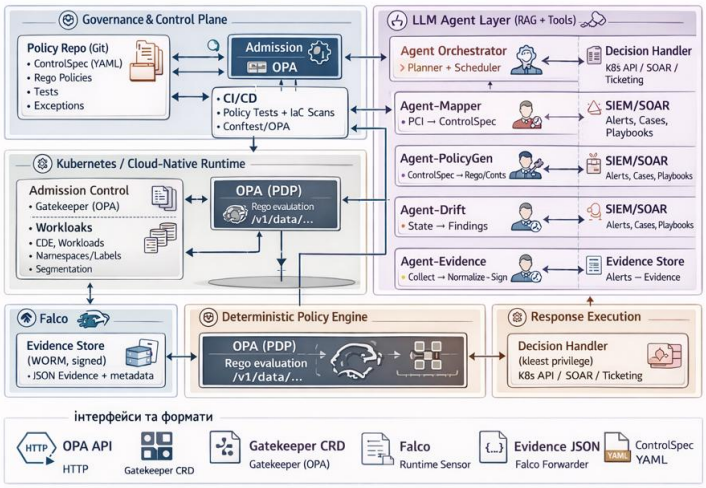

Рис. 1 Референтна архітектура системи автоматизації комплаєнсу на основі Policy-as-Code, OPA, Kubernetes runtime-моніторингу та автономних LLM-агентів

1) OPA API (PDP): HTTP інтерфейс /v1/data/<package>/<rule> для оцінювання Rego-політик над JSON-вхідними даними (admission review, IaC, runtime event). Концептуально це випливає з моделі OPA як policy evaluation engine і характеру Rego як мови політик над структурованими даними [2, 9];

2) Gatekeeper CRD: ConstraintTemplate + Constraint як механізм декларативних обмежень для ресурсів Kubernetes і застосування політик на рівні admission webhooks. [10, 15];

3) Falco event schema: Falco генерує події (алерти) про аномальну поведінку; Falcosidekick пересилає їх через HTTP до зовнішніх систем, може додавати поля, фільтрувати та експортувати метрики [12, 13, 28];

4) Evidence JSON: нормалізована схема, що включає:

– control_id, pci_ref, timestamp, environment, source_system;
– artifact_type (scan_result, config_snapshot, log_digest, attestation);
– hashes (SHA-256), signature (KMS/PKI), retention;
– raw і normalized поля для аудиту.

5) ControlSpec YAML: «контроль як код», який задає, які політики, де і як застосовуються, і які докази збираються.

Порівняння агентів для різних класів LLM. Для PCI DSS критичною є керованість (governance), безпечність інтеграції з інструментами та можливість розгортання у середовищах з обмеженнями на передачу даних (особливо якщо існує ризик контакту з чутливими артефактами). Наявні ризики prompt injection і вимоги до керування AI-

ризиками підкреслюють необхідність розглядати LLM не лише як «якість відповіді», а як компонент з власним профілем ризику [25, 27, 31].

Таблиця 2

Порівняння характеристик агентів на базі різних класів LLM

Критерій	Хмарні закриті LLM (приклад: комерційні API)	Приватні/суверенні LLM (on-prem, керовані)	Open-weight LLM (самохостинг)
Контроль над даними	Високі вимоги до DLP/контрактів; ризик передачі метаданих і промптів	Кращий контроль, але залежить від платформи	Максимальний контроль, якщо інфраструктура зріла
Вартість володіння	OPEX, часто залежить від токенів/викликів	Компроміс між OPEX та CAPEX	CAPEX + експлуатаційні витрати
Якість для нормативних текстів	Зазвичай висока, але потрібен RAG і валідація	Висока/середня залежно від моделі	Середня/висока залежно від tune та контексту
Стійкість до prompt injection	Необхідні системні контрзаходи незалежно від моделі	Аналогічно, потрібні контрзаходи	Аналогічно; також ризики інтеграції сторонніх ваг
Інтеграція із tooling	Зручні SDK/функції, але потрібні «guardrails»	Можливі обмеження інтеграцій	Повна гнучкість, але більше інженерії
Рекомендована роль у PCI DSS	Генерація чернеток ControlSpec/політик, аналітика, звітність (під контролем)	Те саме, плюс аналіз приватних артефактів у межах контуру	Обмежені/критичні контури (evidence, конфігурації) за наявності ресурсів

Реалізація, експеримент і результати.

Реалізація прототипу передбачає поетапне розгортання PaC/SaC інструментів у cloud-native контурі з наступною інтеграцією агентного шару.

Крок 1. Нормативний шар і scoping. Визначається «CDE-подібний сегмент» у Kubernetes – наприклад, через namespaces/labels та мережеву сегментацію. Необхідність коректного scoping і вказівки на значущість сегментації для області застосування підкреслюється в PCI DSS v4.0.1, де відображені міркування щодо scoping і залежність адекватності сегментації від конкретної реалізації та контексту [1, 8].

Крок 2. OPA/Gatekeeper як превентивний контроль. OPA використовується як policy evaluation engine з Rego, а Gatekeeper – як admission webhook для Kubernetes, що застосовує constraints для створення/оновлення ресурсів [9, 10]. Цей крок відповідає логіці «Apply Secure Configurations to All System Components» та «Protect Cardholder Data with Strong Cryptography During Transmission…» у складі принципових вимог PCI DSS, які можна частково відобразити через нормативи конфігурацій (TLS на ingress, заборона privileged, обмеження hostPath) [1, 18].

Крок 3. Falco як runtime детектор. Falco розгортається як runtime security сенсор, що виявляє аномальну поведінку у хостах/контейнерах/Kubernetes [4, 12]. Falcosidekick використовується як проксі для forwarding алертів через HTTP до OPA/інших систем, підтримуючи фільтрацію, додаткові поля та інтеграції [13, 19].

Крок 4. Decision handler. У відповідності до принципу мінімальних привілеїв і розподілу ролей, виконання дій (ізоляція pod, створення incident/ticket, ініціація дій SOAR) виноситься в окремий сервіс з чітко обмеженими правами, а OPA лишається PDP без необхідності доступу до Kubernetes API. Такий підхід описується як доцільний у моделях інтеграції Falco↔OPA через проміжний сервіс.

Крок 5. Агентний шар. Оркестратор агентів виконує:

- парсинг PCI DSS вимог у ControlSpec (Agent-Mapper);
- генерацію політик і тестів (Agent-PolicyGen);
- збір evidence та нормалізацію (Agent-Evidence);
- аналіз дрейфу (Agent-Drift) і підготовку PR (Agent-Remediate);
- формування ROC/AOC-чернеток (Agent-Audit).

З огляду на загальні патерни LLM-агентів (планування, пам’ять, використання інструментів) доцільно використовувати RAG для підкріплення генерації нормативним контентом і локальними політиками/винятками, а також ReAct-подібні стратегії для прозорості «reason+act» траєкторій [15, 28, 29].

Приклади правил Policy-as-Code. Наведені приклади слугують демонстраційними фрагментами для превентивного контролю конфігурацій (admission) та політичної класифікації runtime-подій. Вони не є вичерпним покриттям PCI DSS, а показують, як окремі принципові вимоги (наприклад, безпечні конфігурації, шифрування під час передачі, контроль доступу, журналювання/моніторинг) можуть бути частково формалізовані [9, 10, 12].

```
package pci.k8s.baseline

# Deny privileged containers (Secure Configurations / Least Privilege)
deny[msg] {
  input.review.kind.kind == "Pod"
  c := input.review.object.spec.containers[_]
  c.securityContext.privileged == true
  msg := sprintf("Privileged container is not allowed: %s", [c.name])
}

# Deny hostPath volumes (reduce escape / hardening)
deny[msg] {
  input.review.kind.kind == "Pod"
  v := input.review.object.spec.volumes[_]
  v.hostPath
  msg := sprintf("hostPath volume is not allowed: %s", [v.name])
}
package pci.k8s.ingress_tls

# Ensure TLS enabled on Ingress that exposes services in CDE namespaces
deny[msg] {
  input.review.kind.kind == "Ingress"
  ns := input.review.object.metadata.namespace
  startswith(ns, "cde-")
  not ingress_has_tls(input.review.object.spec)
  msg := sprintf("Ingress in %s must enforce TLS", [ns])
}

ingress_has_tls(spec) {
  spec.tls
  count(spec.tls) > 0
}
package pci.k8s.image_sources
```

```
# Allow images only from approved registries (supports secure development
& deployment governance)
approved_registries := {"registry.corp.example",
"registry.security.example"}

deny[msg] {
  input.review.kind.kind == "Pod"
  c := input.review.object.spec.containers[_]
  img := c.image
  not image_from_approved_registry(img)
  msg := sprintf("Image registry not approved: %s", [img])
}

image_from_approved_registry(img) {
  parts := split(img, "/")
  registry := parts[0]
  approved_registries[registry]
}
package pci.runtime.exec_monitor

# Classify Falco events: interactive shell in CDE workloads
# Expected input: Falco alert forwarded as JSON
default decision := {"action": "allow", "severity": "info", "reason": "no
match"}

decision := {"action": "isolate_pod", "severity": "high", "reason":
reason} {
  input.source == "falco"
  input.k8s.ns != null
  startswith(input.k8s.ns, "cde-")
  input.rule contains "Terminal shell"
  reason := sprintf("Interactive shell detected in CDE namespace: %s",
[input.k8s.ns])
}
package pci.audit.log_monitoring

# Simple evidence policy: require daily log monitoring evidence object
default compliant := false

compliant {
  input.control_id == "PCI-10-daily-log-monitoring"
  input.artifact_type == "log_digest"
  input.period == "daily"
  input.digest != ""
}
```

Сценарії автоматизованого реагування агентів. Сценарії базуються на розділенні ролей «sensor → decision → action» та повинні забезпечувати трасованість і мінімальні привілеї, що корелює з архітектурними підходами інтеграції Falco й OPA та практиками SOAR [13, 21].

Сценарій 1: runtime-shell у CDE. Falco детектує «interactive shell» у namespace cde-payments → Falcosidekick надсилає подію в OPA → OPA повертає isolate_pod →

decision handler застосовує NetworkPolicy «deny all» для pod/namespace або переводить pod у «quarantine» (label + policy), створює incident у SOAR та додає подію в evidence store як «інцидентна перевірка» для вимог моніторингу [12, 13, 28].

Сценарій 2: небезпечний manifest у CI/CD. Agent-PolicyGen додає або оновлює Rego-політики в репозиторії; CI запускає unit tests політик і Conftest-скан на Kubernetes manifests/IaC. Якщо політика блокує, pipeline зупиняється, а Agent-Remediate формує PR із виправленням, додаючи пояснення та посилання на відповідний ControlSpec. Цей процес формалізує «secure configurations» як контроль змін у DevSecOps-циклі [9, 10, 16].

Сценарій 3: витік секретів у репозиторії. Статичний сканер або агент знаходить hardcoded credentials → створюється тикет, ініціюється ротація ключів, а Agent-Audit додає доказ виконання процедури реагування. Релевантність проблеми неконтрольованих секретів у коді показана в дослідженнях [22, 37].

Метрики оцінки відповідності. Метрики групуються в чотири класи:

1) Coverage/traceability: частка вимог PCI DSS, для яких існують ControlSpec і автоматизовані перевірки; частка вимог із прив’язаними доказами. (Прив’язка до структури 12 принципових вимог PCI DSS v4.0.1) [1, 18];

2) Ефективність контролів: частота спрацювань політик, частка істинних/хибних спрацювань для подій Falco, рівень дрейфу конфігурацій. (Falco як runtime detector) [4, 12];

3) Операційні метрики: MTTD/MTTR для інцидентів, тривалість підготовки доказів до аудиту, кількість ручних кроків. (SOAR-підхід як автоматизація управління інцидентами) [21, 22];

4) Безпека агентів: відсоток заблокованих небезпечних tool calls, успішність red-team prompt-injection тестів, відповідність AI-ризик-рамці (AI RMF) [25, 27].

Експериментальний план валідації. План валідації пропонує порівняти три режими:

(A) ручне впровадження контролів;

(B) SaC без LLM-агентів (PaC + runtime + evidence автоматизація без генерації);

(C) SaC + LLM-агенти з детермінованим policy-first виконанням.

Середовище експерименту (незадане як факт; описується як дизайн):

– Kubernetes-кластер (тестовий), сегментований «CDE-like» namespace;
– пайплайн CI/CD із policy tests;
– Falco + Falcosidekick;
– evidence store (append-only);
– набір «порушень» (misconfig corpus): privileged pods, Ingress без TLS, hostPath, images з несанкціонованих registry, runtime exec shell;
– набір «аудитних запитів» (audit queries): довести наявність політик, лог-моніторинг, журналювання.

Опора на методичні рекомендації з контейнерної безпеки (NIST SP 800-190) дозволяє сформувати типові загрози/контрзаходи для контейнерних середовищ, які можна використовувати як основу для misconfig corpus та чеків [7]. Для загальної контрольної структури управління безпековими контролями може використовуватись каталог NIST SP 800-53 як еталон для побудови контрольних тестів і процедур оцінювання (хоча PCI DSS є окремим стандартом) [8].

Гіпотези (підлягають перевірці):

H1: SaC з PaC (OPA/Gatekeeper) зменшує кількість некомплаєнтних конфігурацій, що доходять до runtime, порівняно з ручним процесом [3, 10, 11].

H2: Додавання Falco підвищує покриття «runtime-порушень», які не відловлюються admission-контролем [4, 12].

H3: LLM-агенти з RAG та детермінованими політиками зменшують час підготовки evidence пакета і підвищують трасованість [23, 28, 29].

H4: Без спеціальних контрзаходів prompt injection суттєво знижує безпечність агентів; застосування рекомендацій OWASP і AI RMF зменшує ризики [5, 25, 26, 27].

Результати (стан на момент написання): як безпосередні емпіричні вимірювання в конкретному CDE-середовищі у цій роботі не наводяться через незаданість середовища виконання і недоступність реального контуру оцінювання у межах статті (див. «Обмеження та незадані параметри»). Натомість, наведено:

а) результат синтезу архітектури з літератури;

б) формальні артефакти (ControlSpec-концепт, мермайд-діаграми, Rego-правила);

в) експериментальний дизайн. Додатково використано публікації, які демонструють релевантні компоненти: інтеграційні моделі Falco↔OPA в Kubernetes та аналіз Agentic AI-ризиків у SDLC.

Обговорення та обмеження. Запропонований підхід відповідає логіці PCI DSS v4.0.1, оскільки:

- Зменшує розрив між «вимогою» та «впровадженням» через ControlSpec і PaC/SaC артефакти, що дозволяє будувати повторювані процедури тестування і збір доказів [1, 8].
- Підсилює превентивний контроль через Kubernetes admission (Gatekeeper), що підвищує ймовірність виконання «secure configuration» вимог до моменту розгортання [3, 10, 11].
- Додає runtime спостереження (Falco), що важливо для вимог моніторингу/виявлення порушень, які можуть обійти або не потрапити в admission-шар [4, 12].
- Створює основу для автоматизації інцидент-менеджменту та реакцій за SOAR-подібними принципами, які показані у хмарних дослідженнях [21, 22].

У перспективі «future-dated requirements» PCI DSS v4.x збільшує цінність автоматизації: SaC дозволяє швидше масштабувати нові контролі як код у репозиторіях і pipeline-ах, зберігаючи аудитопридатність через версіонування й тестування [4, 10].

ВИСНОВКИ ТА ПЕРСПЕКТИВИ ПОДАЛЬШИХ ДОСЛІДЖЕНЬ

Стандарт PCI DSS v4.0.1 визначає відповідність як безперервний, орієнтований на доказову базу процес, що поєднує технічні засоби захисту, операційні процедури та систематичні механізми перевірки. У cloud-native середовищах це вимагає переходу від статичної відповідності, що забезпечується на етапі проєктування, до безперервної відповідності, інтегрованої в повний життєвий цикл системи. Засоби контролю мають бути реалізовані у відтворюваний, детермінований і аудиторсько перевірюваний спосіб, що дозволяє оцінювати кожну зміну інфраструктури, подію розгортання та поведінку системи під час виконання на відповідність встановленим вимогам безпеки.

У роботі запропоновано архітектуру Security-as-Code, яка забезпечує реалізацію відповідності шляхом використання декларативних, машинно-перевірюваних політик безпеки та автоматизованого моніторингу поведінки системи. Превентивний контроль реалізується за допомогою підходу Policy-as-Code, який дозволяє перевіряти конфігурації та блокувати зміни, що порушують вимоги безпеки, до їх впровадження. Додатково, механізми runtime-моніторингу забезпечують безперервне виявлення

аномальної або несанкціонованої поведінки після розгортання системи. Інтеграція механізмів автоматизованого прийняття рішень дозволяє виконувати контрольовані реакції відповідно до визначених політик, забезпечуючи цілісність системи та зменшуючи залежність від ручного втручання.

Ключовим науковим результатом роботи є формалізація процесу трансформації нормативних вимог у структуровані специфікації контролів та виконувані політики безпеки. Такий підхід забезпечує прямий зв'язок між високорівневими вимогами відповідності та їх технічною реалізацією, що дозволяє автоматизувати процес формування доказів відповідності та забезпечити їх безперервну актуальність. У результаті перевірка відповідності стає невід'ємною властивістю функціонування системи, а не періодичною процедурою аудиту.

Додатковим внеском роботи є інтеграція автономних агентів на базі великих мовних моделей як інтелектуального шару оркестрації, який підтримує інтерпретацію нормативних вимог, формування специфікацій контролів, аналіз доказів відповідності та підготовку структурованої звітності. При цьому запропонована архітектура базується на принципі policy-first, згідно з яким детермінований механізм виконання політик залишається єдиним авторитетним джерелом рішень у сфері безпеки. Компоненти штучного інтелекту функціонують у межах визначених політик і структурованих джерел знань, що дозволяє підвищити рівень автоматизації та ефективності без зниження надійності та передбачуваності системи.

Запропонована архітектура забезпечує підвищення масштабованості, адаптивності та стійкості процесів забезпечення відповідності у динамічних cloud-native середовищах. На відміну від традиційних підходів, що базуються на періодичних ручних перевірках, запропонована модель забезпечує безперервну перевірку відповідності протягом усього життєвого циклу системи. Це дозволяє гарантувати постійне узгодження поточного стану інфраструктури з вимогами стандартів безпеки навіть за умов частих змін та автоматизованих процесів розгортання.

З практичної точки зору, запропонований підхід формує основу для реалізації безперервного, автоматизованого та аудиторсько перевірюваного забезпечення відповідності у cloud-native інфраструктурах, зокрема на базі Kubernetes. Він дозволяє інтегрувати механізми забезпечення відповідності безпосередньо у процеси DevOps та DevSecOps, перетворюючи відповідність на вбудовану властивість життєвого циклу системи.

Подальші дослідження мають бути спрямовані на експериментальну перевірку ефективності запропонованої архітектури в реальних production-середовищах, включаючи кількісну оцінку повноти контролів, точності виявлення порушень та впливу на продуктивність системи. Також необхідним є розвиток стандартизованих моделей специфікацій контролів, формалізованих схем доказів відповідності та універсальних механізмів автоматизованої звітності. Окрему увагу слід приділити оцінці надійності, безпечності та передбачуваності використання агентів штучного інтелекту в процесах забезпечення відповідності у критично важливих системах.

Загалом результати дослідження підтверджують, що поєднання підходу Security-as-Code, формалізованого виконання політик безпеки та контрольованої інтеграції штучного інтелекту забезпечує ефективну, масштабовану та надійну основу для реалізації безперервної відповідності сучасним стандартам безпеки у cloud-native системах.

СПИСОК ВИКОРИСТАНИХ ДЖЕРЕЛ

1. Chornii, V., Martseniuk, Y., Partyka, A., & Harasymchuk, O. (2025). Information security risks associated with the uncontrolled storage of secrets in source code. *CEUR Workshop Proceedings, 4042*. https://ceur-ws.org/Vol-4042/paper19.pdf
2. Das, B. K. S., & Chu, V. (2023). *Security as code: DevSecOps patterns with AWS*. O'Reilly Media.
3. Kubernetes Blog. (2019). OPA Gatekeeper: Policy and governance for Kubernetes. https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/
4. Lewis, P., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In *Advances in Neural Information Processing Systems (NeurIPS)*. https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf
5. Martseniuk, Y., Partyka, A., Harasymchuk, O., & Korshun, N. (2024). Automated conformity verification concept for cloud security. *CEUR Workshop Proceedings, 3654*. https://ceur-ws.org/Vol-3654/paper3.pdf
6. Martseniuk, Y., et al. (2024). Shadow IT risk analysis in public cloud infrastructure. *CEUR Workshop Proceedings, 3800*. https://ceur-ws.org/Vol-3800/paper3.pdf
7. Mazzola, F., et al. (2023). Runtime security enforcement in containerized environments using Falco. *CEUR Workshop Proceedings, 3421*. https://ceur-ws.org/Vol-3421/
8. National Institute of Standards and Technology. (2020). *Security and privacy controls for information systems and organizations (SP 800-53 Rev. 5)*. https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf
9. National Institute of Standards and Technology. (2023). *Artificial intelligence risk management framework (AI RMF 1.0)*. https://nvlpubs.nist.gov/nistpubs/ai/nist.ai.100-1.pdf
10. Open Policy Agent. (n.d.). *Policy language (Rego)*. https://openpolicyagent.org/docs/policy-language
11. Open Policy Agent Gatekeeper. (n.d.). *Gatekeeper documentation*. https://open-policy-agent.github.io/gatekeeper/website/docs/
12. OWASP Foundation. (n.d.). *LLM prompt injection prevention cheat sheet*. https://cheatsheetseries.owasp.org/cheatsheets/LLM_Prompt_Injection_Prevention_Cheat_Sheet.html
13. OWASP Foundation. (n.d.). *OWASP top 10 for large language model applications*. https://owasp.org/www-project-top-10-for-large-language-model-applications/
14. PCI Security Standards Council. (2018). *PCI SSC cloud computing guidelines v3*. https://www.pcisecuritystandards.org/pdfs/PCI_SSC_Cloud_Guidelines_v3.pdf
15. PCI Security Standards Council. (2022). *Summary of changes from PCI DSS version 3.2.1 to 4.0*. https://listings.pcisecuritystandards.org/documents/PCI-DSS-v3-2-1-to-v4-0-Summary-of-Changes-r1.pdf
16. PCI Security Standards Council. (2024a). *Payment card industry data security standard: Requirements and testing procedures (v4.0.1)*. https://www.middlebury.edu/sites/default/files/2025-01/PCI-DSS-v4_0_1.pdf
17. PCI Security Standards Council. (2024b). *PCI DSS overview*. https://www.pcisecuritystandards.org/standards/pci-dss/
18. PCI Security Standards Council Blog. (2024a, June 11). Just published: PCI DSS v4.0.1. https://blog.pcisecuritystandards.org/just-published-pci-dss-v4-0-1
19. PCI Security Standards Council Blog. (2024b, August 20). Now is the time for organizations to adopt the future-dated requirements of PCI DSS v4.x. https://blog.pcisecuritystandards.org/now-is-the-time-for-organizations-to-adopt-the-future-dated-requirements-of-pci-dss-v4-x
20. Sapsai, O. S., Martseniuk, Y. V., & Partyka, A. I. (2025). Automate cloud security incident management with a SOAR-based approach. *Cybersecurity: Education, Science, Technique, 7*(2). https://science.lpnu.ua/csn/all-volumes-and-issues/volume-7-number-2-2025/automate-cloud-security-incident-management-soar
21. Souppaya, M., Morello, J., & Scarfone, K. (2017). *Application container security guide (SP 800-190)*. National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-190.pdf
22. The Falco Project. (n.d.). *Falco documentation*. https://falco.org/docs/
23. The Falco Project. (n.d.). *Alerts forwarding (Falcosidekick)*. https://falco.org/docs/concepts/outputs/forwarding/
24. Vakhula, O., Kurii, Y., Opirskyy, I., & Susukailo, V. (2024). Security-as-code concept for fulfilling ISO/IEC 27001:2022 requirements. *CEUR Workshop Proceedings, 3654*. https://ceur-ws.org/Vol-3654/paper6.pdf

25. Vakhula, O., & Opirskyy, I. (2024). Research on security-as-code approach for cloud-native applications based on Kubernetes cluster. *CEUR Workshop Proceedings, 3800*. https://ceur-ws.org/Vol-3800/paper6.pdf
26. Vakhula, O., Opirskyy, I., Vorobets, P., Bobko, O., & Kulinich, O. (2025). Research on policy-as-code for implementation of role-based and attribute-based access control. *CEUR Workshop Proceedings, 3991*. https://ceur-ws.org/Vol-3991/paper11.pdf
27. Wang, L., Ma, Y., Zhang, Q., et al. (2024). A survey on large language model-based autonomous agents. *Frontiers of Computer Science*. https://arxiv.org/abs/2308.11432
28. Wei, H., Madhavji, N., & Steinbacher, J. (2025). Understanding everything as code: A taxonomy and conceptual model. *arXiv*. https://arxiv.org/pdf/2507.05100
29. Yao, S., et al. (2022). ReAct: Synergizing reasoning and acting in language models. *arXiv*. https://arxiv.org/abs/2210.03629

КІБЕРБЕЗПЕКА: освіта, наука, техніка
CYBERSECURITY: EDUCATION, SCIENCE, TECHNIQUE
№ 4 (32), 2026
ISSN 2663 – 4023

Oleksandr Vakhula
PhD Student,assistant of Department of Information Protection
Lviv Polytechnic National University, Lviv, Ukraine
ORCID: 0009-0008-5367-3344
oleksandr.p.vakhula@lpnu.ua

Dmytro Marchuk
Student of the Department of Information Protection
Lviv Polytechnic National University, Lviv, Ukraine
ORCID: 0009-0007-7385-8752
dmytro.marchuk.kb.2024@gmail.com

SECURITY-AS-CODE APPROACH FOR AUTOMATING PCI DSS COMPLIANCE USING AUTONOMOUS AGENTS BASED ON LARGE LANGUAGE MODELS

Abstract. This paper investigates the Security-as-Code approach for automating PCI DSS compliance in cloud-native environments using autonomous agents based on large language models. Based on an analysis of PCI DSS v4.0.1, which defines twelve principal requirements and emphasizes continuous assessment, evidence-based validation of controls, and proper scoping, a reference architecture is proposed that integrates declarative security policies, preventive configuration enforcement mechanisms, and runtime behavioral monitoring. The study formalizes a mechanism for transforming regulatory requirements into machine-verifiable policies through the use of control specifications (ControlSpec), a structured knowledge base, and a Retrieval-Augmented Generation approach, ensuring policy provenance and reducing the risk of incorrect generation. A model of autonomous agents is proposed to perform functions including requirements-to-policy transformation, configuration drift analysis, evidence collection and normalization, and the generation of remediation recommendations using controlled change mechanisms. Metrics for evaluating continuous compliance are defined, including control coverage, drift detection latency, Mean Time to Detect (MTTD), Mean Time to Respond (MTTR), completeness of evidence, and detection accuracy. An experimental validation plan is proposed using a test environment that simulates a Cardholder Data Environment. Particular attention is given to the analysis of risks associated with autonomous agents, including model hallucinations, prompt injection attacks, sensitive data leakage, and excessive tool privileges. Mitigation measures are defined, including deterministic policy enforcement, tool isolation, agent action logging, and the use of a human-in-the-loop approach for critical operations. The proposed approach enables continuous, verifiable, and scalable PCI DSS compliance in cloud-native environments.

Keywords: PCI DSS; Security-as-Code; Policy-as-Code; Compliance-as-Code; autonomous agents; large language models; DevSecOps; admission control; continuous compliance.

REFERENCES (TRANSLATED AND TRANSLITERATED)

1. Chornii, V., Martseniuk, Y., Partyka, A., & Harasymchuk, O. (2025). Information security risks associated with the uncontrolled storage of secrets in source code. CEUR Workshop Proceedings, 4042. https://ceur-ws.org/Vol-4042/paper19.pdf
2. Das, B. K. S., & Chu, V. (2023). Security as code: DevSecOps patterns with AWS. O'Reilly Media.
3. Kubernetes Blog. (2019). OPA Gatekeeper: Policy and governance for Kubernetes. https://kubernetes.io/blog/2019/08/06/opa-gatekeeper-policy-and-governance-for-kubernetes/
4. Lewis, P., et al. (2020). Retrieval-augmented generation for knowledge-intensive NLP tasks. In Advances in Neural Information Processing Systems (NeurIPS). https://proceedings.neurips.cc/paper/2020/file/6b493230205f780e1bc26945df7481e5-Paper.pdf
5. Martseniuk, Y., Partyka, A., Harasymchuk, O., & Korshun, N. (2024). Automated conformity verification concept for cloud security. CEUR Workshop Proceedings, 3654. https://ceur-ws.org/Vol-3654/paper3.pdf
6. Martseniuk, Y., et al. (2024). Shadow IT risk analysis in public cloud infrastructure. CEUR Workshop Proceedings, 3800. https://ceur-ws.org/Vol-3800/paper3.pdf
7. Mazzola, F., et al. (2023). Runtime security enforcement in containerized environments using Falco. CEUR Workshop Proceedings, 3421. https://ceur-ws.org/Vol-3421/

8. National Institute of Standards and Technology. (2020). Security and privacy controls for information systems and organizations (SP 800-53 Rev. 5). https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-53r5.pdf
9. National Institute of Standards and Technology. (2023). Artificial intelligence risk management framework (AI RMF 1.0). https://nvlpubs.nist.gov/nistpubs/ai/nist.ai.100-1.pdf
10. Open Policy Agent. (n.d.). Policy language (Rego). https://openpolicyagent.org/docs/policy-language
11. Open Policy Agent Gatekeeper. (n.d.). Gatekeeper documentation. https://open-policy-agent.github.io/gatekeeper/website/docs/
12. OWASP Foundation. (n.d.). LLM prompt injection prevention cheat sheet. https://cheatsheetseries.owasp.org/cheatsheets/LLM_Prompt_Injection_Prevention_Cheat_Sheet.html
13. OWASP Foundation. (n.d.). OWASP top 10 for large language model applications. https://owasp.org/www-project-top-10-for-large-language-model-applications/
14. PCI Security Standards Council. (2018). PCI SSC cloud computing guidelines v3. https://www.pcisecuritystandards.org/pdfs/PCI_SSC_Cloud_Guidelines_v3.pdf
15. PCI Security Standards Council. (2022). Summary of changes from PCI DSS version 3.2.1 to 4.0. https://listings.pcisecuritystandards.org/documents/PCI-DSS-v3-2-1-to-v4-0-Summary-of-Changes-r1.pdf
16. PCI Security Standards Council. (2024a). Payment card industry data security standard: Requirements and testing procedures (v4.0.1). https://www.middlebury.edu/sites/default/files/2025-01/PCI-DSS-v4_0_1.pdf
17. PCI Security Standards Council. (2024b). PCI DSS overview. https://www.pcisecuritystandards.org/standards/pci-dss/
18. PCI Security Standards Council Blog. (2024a, June 11). Just published: PCI DSS v4.0.1. https://blog.pcisecuritystandards.org/just-published-pci-dss-v4-0-1
19. PCI Security Standards Council Blog. (2024b, August 20). Now is the time for organizations to adopt the future-dated requirements of PCI DSS v4.x. https://blog.pcisecuritystandards.org/now-is-the-time-for-organizations-to-adopt-the-future-dated-requirements-of-pci-dss-v4-x
20. Sapsai, O. S., Martseniuk, Y. V., & Partyka, A. I. (2025). Automate cloud security incident management with a SOAR-based approach. Cybersecurity: Education, Science, Technique, 7(2). https://science.lpnu.ua/csn/all-volumes-and-issues/volume-7-number-2-2025/automate-cloud-security-incident-management-soar
21. Souppaya, M., Morello, J., & Scarfone, K. (2017). Application container security guide (SP 800-190). National Institute of Standards and Technology. https://nvlpubs.nist.gov/nistpubs/specialpublications/nist.sp.800-190.pdf
22. The Falco Project. (n.d.). Falco documentation. https://falco.org/docs/
23. The Falco Project. (n.d.). Alerts forwarding (Falcosidekick). https://falco.org/docs/concepts/outputs/forwarding/
24. Vakhula, O., Kurii, Y., Opirskyy, I., & Susukailo, V. (2024). Security-as-code concept for fulfilling ISO/IEC 27001:2022 requirements. CEUR Workshop Proceedings, 3654. https://ceur-ws.org/Vol-3654/paper6.pdf
25. Vakhula, O., & Opirskyy, I. (2024). Research on security-as-code approach for cloud-native applications based on Kubernetes cluster. CEUR Workshop Proceedings, 3800. https://ceur-ws.org/Vol-3800/paper6.pdf
26. Vakhula, O., Opirskyy, I., Vorobets, P., Bobko, O., & Kulinich, O. (2025). Research on policy-as-code for implementation of role-based and attribute-based access control. CEUR Workshop Proceedings, 3991. https://ceur-ws.org/Vol-3991/paper11.pdf
27. Wang, L., Ma, Y., Zhang, Q., et al. (2024). A survey on large language model-based autonomous agents. Frontiers of Computer Science. https://arxiv.org/abs/2308.11432
28. Wei, H., Madhavji, N., & Steinbacher, J. (2025). Understanding everything as code: A taxonomy and conceptual model. arXiv. https://arxiv.org/pdf/2507.05100
29. Yao, S., et al. (2022). ReAct: Synergizing reasoning and acting in language models. arXiv. https://arxiv.org/abs/2210.03629

Отримано редакцією журналу / Received: 05.01.26
Прорецензовано / Revised: 21.02.26
Схвалено до друку / Accepted: 26.03.26