الجمهورية العربية السورية
المعهد العالي للعلوم التطبيقية والتكنولوجيا
قسم النظم المعلوماتية – شبكات ونظم التشغيل
العام الدراسي 2026/2025

**مشروع تخرج**

أعد لنيل درجة الإجازة في هندسة النظم المعلوماتية

# نظام أمني ذكي موزع لكشف التسلل اعتماداً على تقنيات Hadoop وApache Spark

تقديم

أريج حيدر محمد

إشراف

الدكتور حسين خيو
المهندس عمار مخلوف

8/8/2026

# إهداء

**شکر**

# الخلاصة

مع التزايد المستمر في الهجمات السيبرانية واتساع حجم سجلات حركة الشبكة الناتجة عن البيئات السحابية، برزت الحاجة إلى نظام قادر على استيعاب هذا الكمّ الهائل من البيانات وتحليله في الوقت الفعلي، وهو ما دفع إلى تبنّي نظام أمني ذكي موزّع لكشف التسلل يعتمد في بنيته على تقنيات المعالجة الموزعة والتخزين الكبير ممثّلة بـ Apache Hadoop وApache Spark، مع دمج تقنيات الذكاء الاصطناعي للتمييز بين السلوك الطبيعي والسلوك المشبوه لحركة الشبكة.

اعتمد المشروع منذ بدايته على رؤية هندسية تراعي خصائص التوزّع، وقابلية التوسّع، والقدرة على معالجة كميات كبيرة من السجلات في الوقت الفعلي، وذلك بالفصل بين طبقات النظام الأساسية: طبقة تجميع البيانات، وطبقة التخزين الموزّع، وطبقة التحليل الذكي، وطبقة الاستجابة والتحكم.

شمل التنفيذ بناء بنية تحتية متكاملة تضم عقد تخزين موزّعة عبر HDFS، إضافة إلى محرك معالجة لحظية ذكي عبر Apache Spark يتيح تحليل تدفقات الشبكة فور وصولها واكتشاف الأنماط غير الطبيعية دون تأخير يُذكر.

وفي مرحلة الاختبار، خضع النظام لتجارب متعددة تحاكي بيئة تشغيل حقيقية، بتوليد حركة شبكية من عدة مصادر افتراضية بالتوازي، لقياس قدرة النظام على استقبال البيانات وتحليلها والاستجابة لها دون انقطاع. وقد أظهرت النتائج قدرة النظام على تحقيق دقة عالية في كشف الهجمات مع الحفاظ على استقرار الأداء، مما يؤكد صلاحية البنية المعتمدة للتوسّع مستقبلاً بإضافة موارد معالجة أو تخزين إضافية دون الحاجة لإعادة هيكلة النظام.

يمثّل هذا المشروع تجربة عملية لبناء نظام أمني موزّع يجمع بين مبادئ الحوسبة الكبيرة والذكاء الاصطناعي، مع قابلية واضحة للتكيّف والنمو في بيئات سحابية متغيّرة ومتزايدة التعقيد.



# Abstract

As cyber-attacks continue and network traffic logs within a cloud environment continue to grow, the requirement has arisen to be able to ingest and analyze this massive amount of data in real time. This spurred the implementation of a smart, distributed intrusion detection system (IDS) that leverages distributed processing and large-scale storage technologies, such as Apache Hadoop and Apache Spark, and couples them with artificial intelligence (AI) methods for detecting whether the network is being used in an acceptable manner. The project's engineering vision focused on the properties of distribution, scalability, and ability to treat big data in real time, realized through the separation of the project into its core functional layers – data ingestion, distributed storage, intelligent analysis, response, and control. This implementation included the construction of the whole infrastructure with distributed storage nodes using HDFS, as well as an intelligent real-time processing engine (Apache Spark), which allows analyzing network flows at real time, and detecting anomalous patterns with minimal delay. The system was also tested in several experiments that simulated a realistic operating environment, with network traffic coming from multiple virtual sources that would be processed in parallel, with the aim of assessing how well the system could ingest, analyse and respond to network traffic without any disruption. The results showed that the system can 100% detect with stable performance, which proves that the proposed structure can be scaled up in the future by adding processing or storage resources without changing the structure itself. The project is a practical realization of an application that integrates concepts of big data computing and artificial intelligence and will have plenty of potential for adaptation and expansion in ever more complex and changing cloud settings.

## Contents

**الفصل الأول** ..... **8**

1.1– مقدمة ..... 8

2.1– الهدف من المشروع ..... 8

3.1 المتطلبات. ..... 8

1.3.1- المتطلبات الوظيفية (Functional Requirements) ..... 8

2.3.1- المتطلبات غير الوظيفية (Non Functional Requirements) ..... 9

**الفصل الثاني** ..... **10**

..... **10**

**الفصل الثالث** ..... **11**

..... **11**

**الفصل الرابع** ..... **12**

**ي** ..... **12**

1.4– مخطط حالات الاستخدام ..... 12

2.4– الوصف النصي لكل حالة استخدام مع مخطط تتالي النظام الموافق لها ..... 13

1.2.4- حالة استخدام: تسجيل الدخول ..... 13

2.2.4– حالة استخدام: عرض نظرة عامة على لوحة التحكم ..... 14

3.2.4- حالة استخدام: كشف التسلل وإطلاق تنبيه ..... 16

4.2.4- حالة استخدام: مراجعة تفاصيل التنبيه وتحديد الحكم ..... 17

5.2.4- حالة استخدام: الاستعلام عن سُمعة عنوان IP ..... 19

6.2.4- حالة استخدام: حظر عنوان IP ..... 20

7.2.4- حالة استخدام: رفع الحظر عن عنوانIP ..... 21

8.2.4- حالة استخدام: عرض أداء النموذج ..... 22

9.2.4- حالة استخدام: تفعيل إعادة تدريب النموذج ..... 23

10.2.4- حالة استخدام: تصدير التنبيهات المُراجَعة ..... 25

**الفصل الخامس** ..... **27**

. ..... **27**

1.5– مقدمة ..... 27

2.5– النمط المعماري المعتمد ..... 27

3.5– مكونات النظام ..... 28

3.5- معمارية العنقود الموزع: نمط السيادة والتنفيذ(Master-Worker Cluster Architecture) .. 31
4.5- تصميم نموذج الذكاء الصنعي ..... 33
5.5- الأنماط التصميمة المعتمدة ..... 34
6.5- تصميم التخزين حسب طبيعة البيانات ..... 36
7.5- تصميم الية إعادة التدريب ..... 37
**الفصل السادس ..... 40**
**..... 40**
1.6- تقييم أثر توزيع البيانات على كفاءة التدريب و الإنتاجية ..... 40
2.6- توازي البيانات مقابل توازي النموذج ..... 60
**الفصل السابع ..... 64**
**..... 64**
1.7- بيئة التنفيذ ..... 64
1.1.7- مقدمة ..... 64
2.1.7- البنية التحتية الموزعة للعنقود ..... 64
3.1.7مبدأ الحوسبة بالحاويات (Containerization) ..... 64
4.1.7التنسيق وإدارة الخدمات عبر Docker Compose ..... 65
2.7- طبقات النظام ..... 66
1.2.7 – طبقة التقاط و تجميع السجلات الشبكية ..... 66
2.2.7 – طبقة النقل و البث اللحظي ..... 68
3.2.7 – طبقة المعالجة و التحليل اللحظي ..... 68
4.2.7 – طبقة الخدمة التخزين الدائم و الموزع ..... 70
5.2.7 – طبقة الخدمة الخلفية و التحكم ..... 71
5.2.7 – طبقة العرض ..... 72
3.7 التقنيات و الأدوات المستخدمة ..... 74
**الفصل الثامن ..... 78**
**..... 78**
1.8- الاختبارات الوظيفية ..... 78
1.1.8- اختبارات الوحدة ..... 78
2.8 اختبارات الأداء – اختبار الحمل ..... 79
**المراجع ..... 92**

# قائمة الجداول

الجدول 1- رموز وقيم العناصر المستخدمة في الدارة. **Error! Bookmark not defined. .....**

# الاختصارات

# الرموز

.

# الفصل الأول

# التعريف بالمشروع

*نقدّم في هذا الفصل تصوراً عاماً حول فكرة المشروع وأهدافه الأساسية، والمتطلبات*

## 1.1– مقدمة

تُعد أنظمة كشف التسلل من أبرز أدوات الدفاع في عصر الحوسبة السحابية، حيث تشكّل خط المراقبة الأول القادر على رصد أي نشاط غير طبيعي قبل أن يتحوّل إلى اختراق فعلي. وتأتي حركة السجلات الشبكية في مقدّمة مصادر البيانات المستخدمة في هذا الكشف، لما تحمله من دلالات دقيقة على سلوك المستخدمين والأجهزة داخل الشبكة.

وانطلاقاً من هذا الدور المتزايد لحركة الشبكة في كشف التهديدات، يهدف هذا المشروع إلى بناء نظام أمني ذكي موزّع قادر على تحليل هذه الحركة بشكل لحظي ومستمر، مع مراعاة قدرته على استيعاب الأحمال المتزايدة من السجلات وتحمّل الضغط التشغيلي الناتج عن اتساع البيئة السحابية، بما يضمن استمرارية الكشف واستجابته لمتطلبات النمو المستقبلي في حجم البيانات وعدد المصادر المراقَبة.

## 2.1- الهدف من المشروع

يهدف هذا المشروع إلى بناء نظام أمني متكامل لكشف التسلل يتميّز بالتوزّع وقابلية التوسّع، بحيث يتيح تشغيل كل مكوّن من مكوناته — من تجميع البيانات إلى تخزينها وتحليلها والاستجابة لها — بشكل مستقل، مما يحقّق إدارة فعالة للموارد ويستوعب النمو التدريجي في حجم السجلات وعدد السيرفرات المراقَبة. ويركّز المشروع على تلبية متطلبات الأداء العالي وتحمّل الضغط التشغيلي، وضمان استمرارية الكشف والاستجابة دون انقطاع.

ولتحقيق هذه الأهداف، يعتمد المشروع على نمط معماري موزّع يتيح فصل المهام بين طبقات جمع البيانات والتخزين والمعالجة والاستجابة وتوزيعها على مكوّنات مستقلة، مع دمج أدوات مراقبة وتحليل أداء تمكّن من تتبّع سلوك النظام في الزمن الحقيقي، واكتشاف الأعطال أو نقاط الاختناق بسرعة. ولذلك، تمّ اختيار معمارية الحوسبة الموزّعة القائمة على Apache Hadoop وApache Spark كنمط تصميم رئيسي، مع الاعتماد على Apache Kafka كقناة بث موزّعة بين المكوّنات، لتوفير بيئة قابلة للتوسّع الأفقي، وتسهيل عمليات إضافة موارد معالجة أو تخزين جديدة دون التأثير على الخدمة القائمة، مما يجعل النظام نموذجاً تطبيقياً يعكس متطلبات أنظمة الأمن السيبراني الحديثة في البيئات السحابية.

## 3.1 المتطلبات.

نقسم المتطلبات التي يجب أنّ يحققها النظام إلى متطلبات وظيفية ومتطلبات غير وظيفية.

### 1.3.1- المتطلبات الوظيفية (Functional Requirements)

1. يجب على النظام أن يقوم بجمع السجلات الشبكية بشكل تلقائي ومستمر دون تدخل بشري.

2. يجب على النظام أن يخزّن السجلات الخام ونتائج التحليل في بيئة موزعة (Hadoop HDFS) بحيث لا تضيع أي بيانات حتى في حال تعطّل أحد مكوّنات التخزين.

3. يجب على النظام أن يكون قادراً على تحليل حركة الشبكة على مرحلتين — تمييز الحركة الطبيعية عن المشبوهة أولاً، ثم تحديد نوع الهجوم (كـ Brute Force وDDoS) — باستخدام خوارزميات تعلم الآلة في الوقت الفعلي.

4. يجب على النظام أن يرسل تنبيهاً فورياً إلى لوحة التحكم عند اكتشاف أي نشاط مشبوه، على أن يتضمن التنبيه نوع الهجوم وعنوان IP المصدر ووقت الاكتشاف وعدد مرات التكرار.

5. يجب على النظام أن يوفّر واجهة رسومية للمدير تعرض حالة النظام وإحصائيات الهجمات وقائمة العناوين المحظورة بشكل مباشر ومحدّث.

6. يجب على النظام أن يتيح للمدير مراجعة كل تنبيه والاطلاع على الأدلة التي استند إليها النموذج (الخصائص الشبكية الخام وسمعة عنوان المصدر) قبل إصدار حكم بشأنه.

7. يجب على النظام أن يكون قادراً على حظر عنوان IP المهاجم يدوياً بأمر من المدير، مع تسجيل كل إجراء في قاعدة البيانات.

8. يجب على النظام أن يُشترط تسجيل الدخول بحساب مصرَّح قبل إتاحة الوصول لأي من وظائف لوحة التحكم.

9. يجب على النظام أن يتيح للمدير إطلاق عملية إعادة تدريب للنموذج بالاعتماد على الأحكام التي راجعها، مع عرض حالة العملية ونتيجتها.

### 2.3.1- المتطلبات غير الوظيفية (Non Functional Requirements)

قابلية التوسع: يجب على النظام أن يكون قابلاً للتوسع بإضافة حاويات أو عقد معالجة جديدة دون الحاجة إلى تعديل في الكود أو إيقاف الخدمة.

الأداء: يجب على النظام أن يكتشف الأنماط الهجومية المستمرة (كـ DDoS وBrute Force) خلال دورة معالجة واحدة لا تتجاوز 10 ثانية، وهو ما يتناسب مع الطبيعة المتكررة لهذا النوع من الهجمات.

التوافرية: يجب على النظام أن يستمر في العمل حتى في حال تعطّل إحدى عقد التخزين، مستفيداً من آلية النسخ الاحتياطي في Hadoop.

سهولة الاستخدام: يجب أن تكون لوحة التحكم واضحة وبسيطة بحيث يستطيع أي مدير نظام استخدامها دون تدريب مسبق.

الأمان: يجب على النظام أن لا يسمح بالوصول إلى وظائف لوحة التحكم الحساسة إلا بعد مصادقة ناجحة مع الاعتماد على الشبكة الداخلية للعنقود لحماية الاتصال بين الخدمات.

.

الفصل الثاني

# الدراسة النظرية

*نعرض في هذا الفصل الدراسات النظرية المستخدمة ضمن هذا العمل.*

## 1.2- مجموعة البيانات المعتمدة (Dataset)

اعتُمدت في هذا المشروع مجموعة بيانات CSE-CIC-IDS2018 العالمية لتدريب نموذج الذكاء الصنعي، وهي واحدة من أحدث وأضخم مجموعات البيانات المرجعية في مجال كشف التسلل (Intrusion Detection Systems)، إذ طُوِّرت بالتعاون بين المنظمة الكندية للأمن السيبراني (CIC) ومعهد الأمن السيبراني في جامعة نيو برونزويك (University of New Brunswick)، على بنية تحتية سحابية محاكاة على منصة Amazon AWS.

تمتاز هذه المجموعة بأنها تعكس حركة شبكية واقعية جداً (إذ تم توليدها من تفاعل فعلي بين مئات الأجهزة الافتراضية على مدى عدة أيام متتالية، بدلاً من الاعتماد على بيانات مصطنعة أو مبسّطة كما هو الحال في مجموعات بيانات أقدم مثل KDD99 وNSL-KDD. كما تغطي المجموعة سيناريوهات هجومية متنوعة تشمل هجمات Brute Force، وDoS، وDDoS، وBotnet، وWeb Attacks، إلى جانب حركة مرور طبيعية (Benign)، ما يجعلها مناسبة لبناء نموذج تصنيف قادر على التمييز بين أنواع متعددة من التهديدات وليس نوعاً واحداً فقط.

يحتوي كل سجل (تدفق شبكي) في هذه المجموعة على أكثر من 80 خاصية إحصائية مستخرَجة باستخدام أداة CICFlowMeter، تصف خصائص الاتصال بدقة، كعدد الحزم المرسلة والمستقبَلة، وإجمالي البايتات المنقولة، ومدة الجلسة (Flow Duration)، وأعلام بروتوكول TCP (TCP Flags) كـ SYN وACK وRST. هذا الغنى في عدد الخصائص وتنوعها هو ما مكّن النموذج المدرَّب من التمييز بدقة عالية بين أنماط الحركة الطبيعية وأنماط الهجمات المختلفة، كما سيُعرض لاحقاً في نتائج التدريب والاختبار.

## 2.6مشاكل الداتا سيت

## -2.2

## -3.2

- 

**-4.2**

الفصل الثالث

# الدراسة المرجعية

*يتضمن التقرير ب.*

**1.3- مقدمةتنسيقها.**

**2.3- الأشكال**

**-4.3**

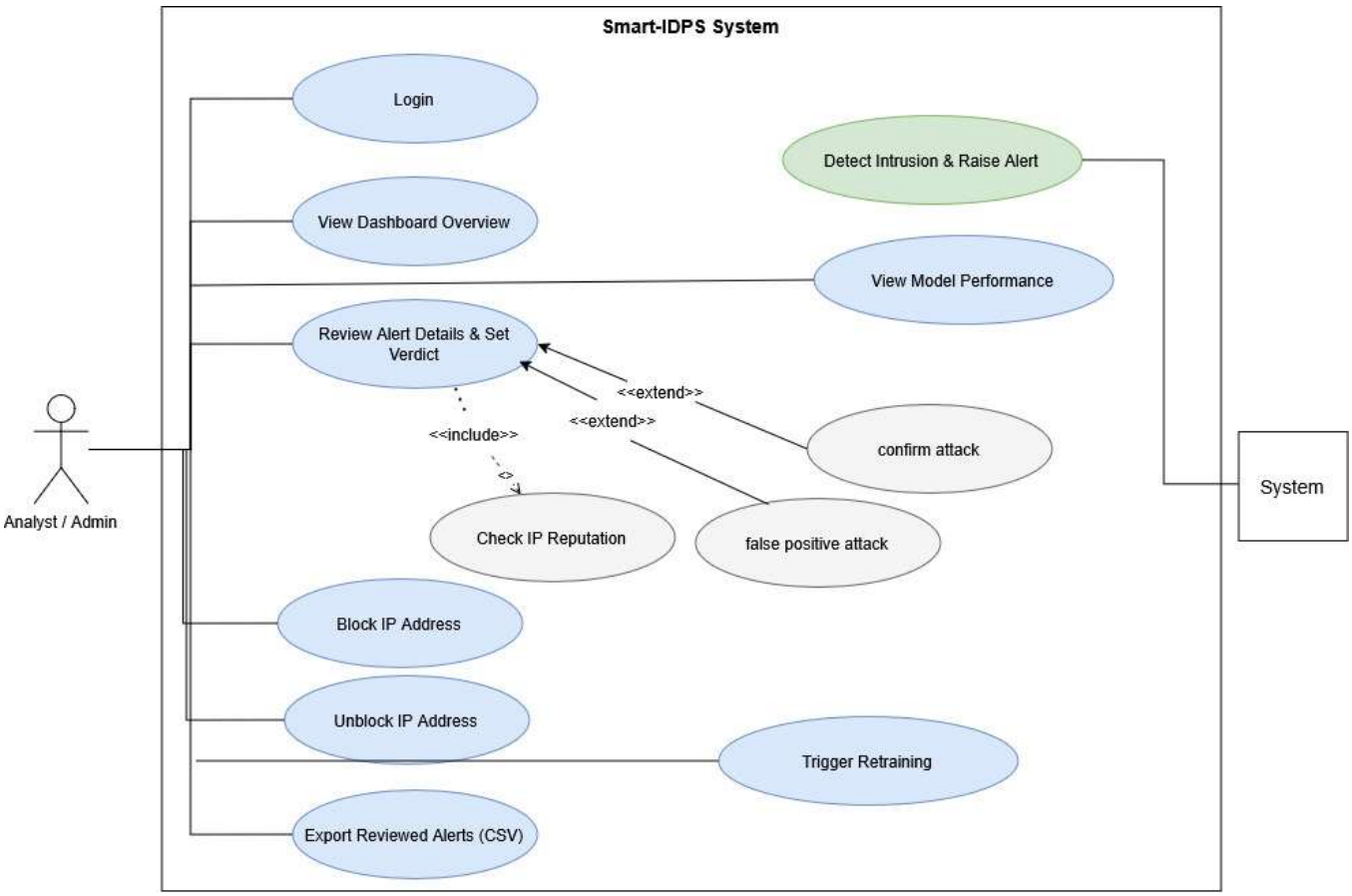
الفصل الرابع

# الدراسة التحليلية

*يوضح هذا الفصل تحليل النظام ،ونورد مخطط حالات الاستخدام مع الوصف النصي لكل حالة و مخطط تتالي النظام*

## 1.4- مخطط حالات الاستخدام

نعرض في هذا الشكل حالات الاستخدام المختلفة ضمن النظام




*الشكل 1: مخطط حالات الاستخدام*

## 2.4- الوصف النصي لكل حالة استخدام مع مخطط تتالي النظام الموافق لها

سنقوم بهذا القسم بسرد نصي لأهم حالات الاستخدام مع مخطط تتالي الاستخدام الموافق لها

جدول 1: حالة استخدام تسجيل الدخول

### 1.2.4- حالة استخدام: تسجيل الدخول

| | |
|---|---|
| اسم الحالة : تسجيل دخول | |
| الوصف (Description) | يقوم الأدمن بتسجيل الدخول إلى لوحة التحكم باستخدام بيانات الاعتماد الخاصة به للوصول إلى وظائف النظام |
| الفاعلين (Actors) | Analyst/Admin |
| الشروط السابقة(Precondition) | لا يوجد |
| الشروط اللاحقة (Postcondition) | تم تسجيل دخول الأدمن بنجاح، وأصبح بحوزته رمز جلسة صالح لاستخدام النظام |

**سير الأحداث (السيناريو الأساسي الناجح):**

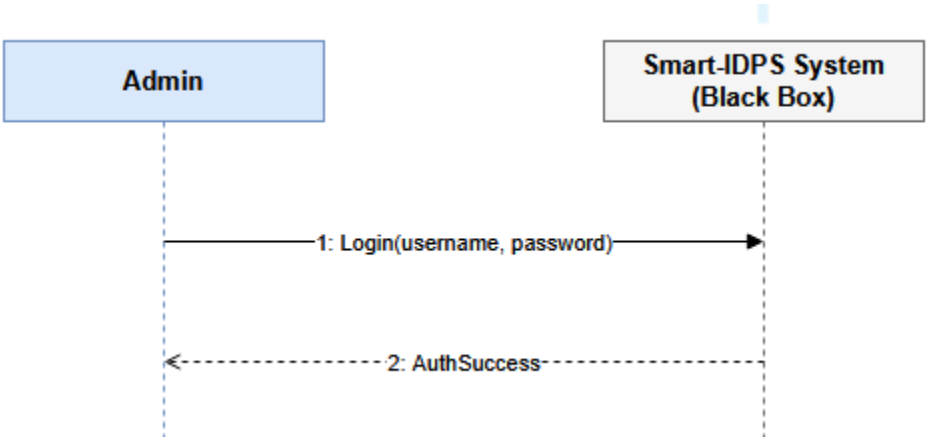
| الأدمن | النظام |
|---|---|
| 1.يطلب تسجيل الدخول. | |
| | 2. يطلب النظام إدخال اسم المستخدم وكلمة المرور. |
| 3. يُدخِل الأدمن بيانات الاعتماد ويطلب تأكيد العملية. | |
| | 4. يتحقق النظام من تطابق البيانات مع الحساب الإداري المعرَّف، ويُصدِر رمز جلسة عشوائياً، ويعيده إلى الأدمن مع تأكيد نجاح الدخول. |

**المسارات البديلة:**
لا يوجد.

**مسارات الأخطاء:**
في حال تحقّقت الحالة التالية في المرحلة رقم 4:

**E1:** بيانات الاعتماد (اسم المستخدم أو كلمة المرور) غير صحيحة.

في هذه الحالة، يُعيد النظام رسالة خطأ توضّح عدم صحة بيانات الدخول، ويطلب من الأدمن إعادة إدخالها من جديد، دون إصدار أي رمز جلسة.




الشكل 2: مخطط تتالي النظام لحالة تسجيل الدخول

### 2.2.4- حالة استخدام: عرض نظرة عامة على لوحة التحكم

| |
|---|
| اسم الحالة : عرض نظرة عامة على لوحة التحكم |

| | |
|---|---|
| الوصف (Description) | يستعرض الأدمن ملخصاً حياً لحالة النظام، يشمل عدد التنبيهات اليومية، توزيع القرارات (مؤكدة/كاذبة/معلَّقة)، عدد العناوين المحظورة، ومؤشر استمرارية تدفق البيانات(Live/Offline) |
| الفاعلين (Actors) | Analyst/Admin |
| الشروط السابقة(Precondition) | الأدمن مسجل الدخول |
| الشروط اللاحقة (Postcondition) | تم عرض الملخص الحي لحالة النظام |

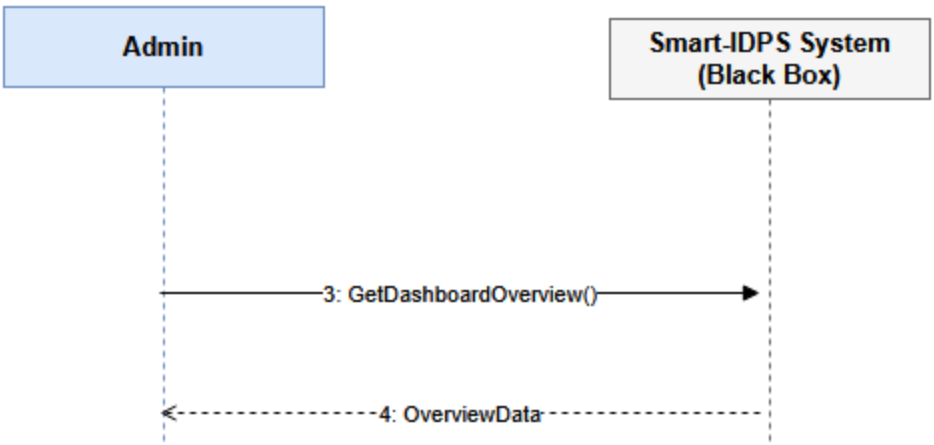
جدول 2 :حالة استخدام عرض نظرة عامة على لوحة التحكم

**سير الأحداث (السيناريو الأساسي الناجح):**

| الأدمن | النظام |
|---|---|
| 1. يفتح صفحة النظرة العامة. | |
| | 2.يستعلم النظام من قاعدة البيانات عن إحصائيات التنبيهات اليومية، وتوزيع القرارات، وعدد العناوين المحظورة. |
| | 3.يحسب النظام مدى حداثة آخر تنبيه وارد، ليحدد ما إذا كان تدفق البيانات حياً حالياً. |
| | 4. يعرض النظام الملخص الكامل على الشاشة. |

المسارات البديلة: لا يوجد.

مسارات الأخطاء: لا يوجد.




الشكل 3: مخطط تتالي النظام لعرض نظرة عامة على لوحة التحكم

### 3.2.4- حالة استخدام: كشف التسلل وإطلاق تنبيه

| | |
|---|---|
| اسم الحالة : كشف تسلل | وإطلاق تنبيه |
| الوصف (Description) | يقوم محرك الذكاء الاصطناعي بتحليل حركة الشبكة الواردة عبر النموذج الهرمي، وعند اكتشاف سلوك هجومي، يُسجّل تنبيهاً جديداً ويُبثّه فوراً إلى لوحة التحكم |
| الفاعلين (Actors) | AI Detection Engineفاعل غير بشري، يبادر تلقائياً |
| الشروط السابقة(Precondition) | يتلقى محرك التحليل دفعة سجلات شبكية جديدة من قناة البث |
| الشروط اللاحقة (Postcondition) | تم تسجيل التنبيه وبثّه إلى جميع الجلسات المفتوحة في لوحة التحكم |

جدول 3:حالة استخدام: كشف التسلل وإطلاق تنبيه

**سير الأحداث (السيناريو الأساسي الناجح):**

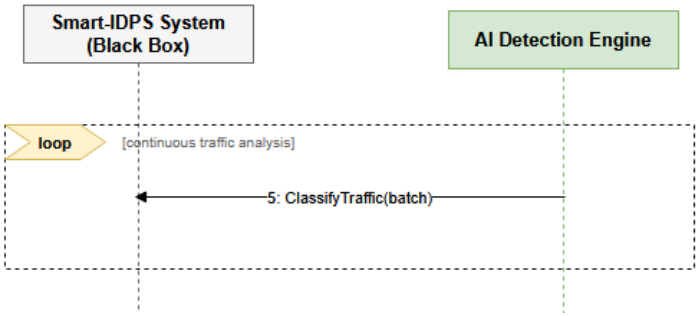
| | |
|---|---|
| محرك الكشف | النظام |
| 1. يُصنِّف دفعة السجلات الواردة عبر المرحلة الأولى (طبيعي/هجوم). | |

| | |
|---|---|
| 2 .لكل سجل مُصنَّف كهجوم، يُحدِّد نوعه عبر المرحلة الثانية. | |
| | 3. يستقبل النظام نتيجة التصنيف، ويتحقق مما إذا كان هجوماً من نفس النوع ومن نفس المصدر قد سُجِّل خلال آخر 60 ثانية. |
| | 4. إن لم يوجد تنبيه مطابق حديث، يُسجِّل النظام تنبيهاً جديداً ويبثّه فوراً عبر القناة الحية إلى جميع الجلسات المفتوحة.. |

المسارات البديلة:

A1: إن وُجد تنبيه مطابق (نفس المصدر ونفس نوع الهجوم) سُجِّل خلال آخر 60 ثانية، يُحدِّث النظام عدد التكرارات لذلك التنبيه بدلاً من إنشاء تنبيه جديد، ويبثّ نسخة محدَّثة منه.

مسارات الأخطاء: لا يوجد.




الشكل 4: مخطط تتالي النظام كشف التسلل وإطلاق تنبيه

### 4.2.4- حالة استخدام: مراجعة تفاصيل التنبيه وتحديد الحكم

| | |
|---|---|
| اسم الحالة : مراجعة | التفاصيل التنبيه وتحديد الحكم |
| الوصف (Description) | يفتح الأدمن تنبيهاً معيّناً للاطلاع على تفاصيله الكاملة، ثم يُصدِر حكمه النهائي بشأنه |
| الفاعلين (Actors) | Analyst/Admin |
| الشروط السابقة(Precondition) | الأدمن مسجل الدخول، ويوجد تنبيه واحد على الأقل بانتظار المراجعة |
| الشروط اللاحقة (Postcondition) | تم تحديث حكم التنبيه في قاعدة البيانات |

جدول 4: حالة الاستخدام مراجعة تفاصيل التنبيه وتحديد الحكم

**سير الأحداث (السيناريو الأساسي الناجح):**

| الأدمن | النظام |
|---|---|
| 1. يختار الأدمن تنبيهاً من قائمة التنبيهات. | |
| | 2. يجلب النظام تفاصيل التنبيه الكاملة، بما فيها الخصائص الشبكية الخام. |
| | 3.يسستعلم النظام تلقائياً عن سُمعة عنوان IP المصدر: *الاستعلام عن سُمعة عنوان IP* ، ويُرفِق النتيجة بالتفاصيل المعروضة. |
| 4.يُقيِّم الأدمن الحالة بناءً على التفاصيل المعروضة، ويُصدِر حكمه. | |
| | 5.يُحدِّث النظام حكم التنبيه في قاعدة البيانات. |

المسارات البديلة:

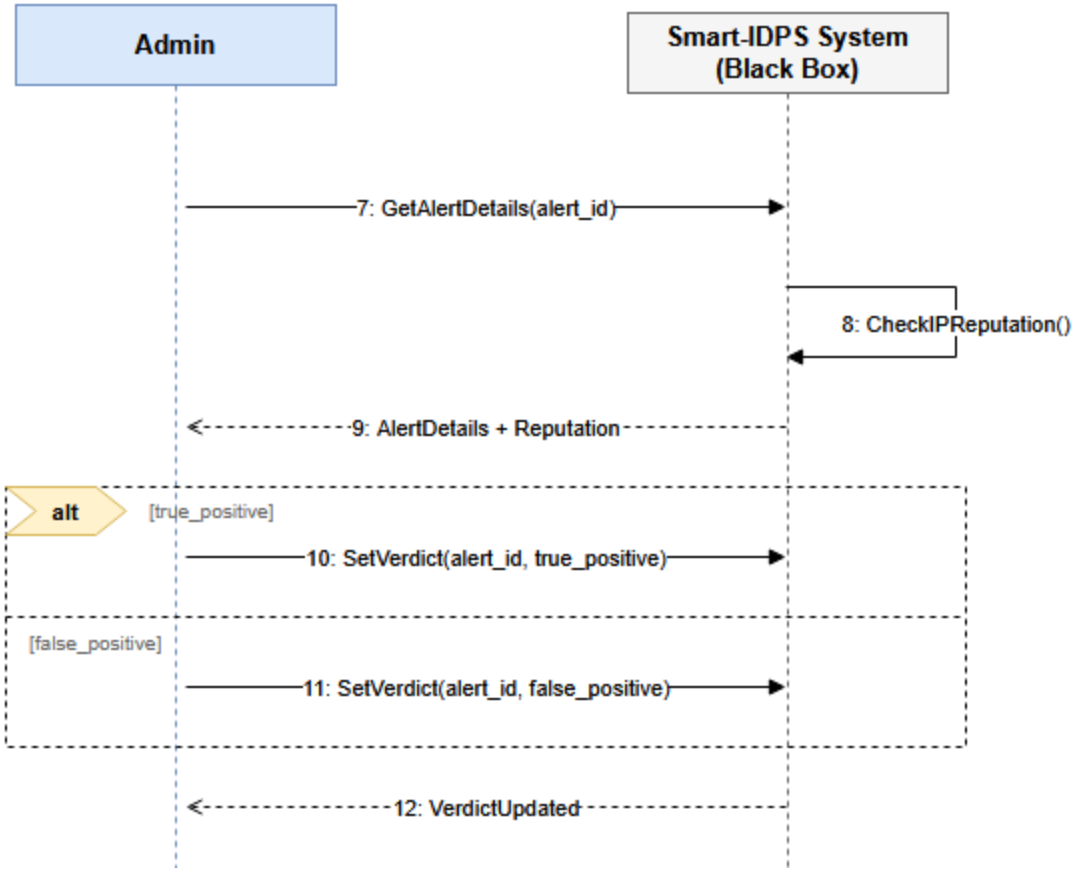
A1 — تأكيد الهجوم: في الخطوة 4، يختار الأدمن تصنيف التنبيه كهجوم حقيقي (true_positive).

A2 — تصنيفه كإنذار كاذب: في الخطوة 4، يختار الأدمن تصنيف التنبيه كإنذار كاذب (false_positive).

مسارات الأخطاء:

E1: في حال أُرسِلت قيمة حكم غير مقبولة، يرفض النظام التحديث ويعيد رسالة خطأ.

ملاحظة تحليلية: يُشكِّل حكم الأدمن في هذه الحالة الحكم المرجعي (Ground Truth) الوحيد المتاح لتقييم أداء النموذج بعد النشر، إذ تُستخدَم تراكمية هذه الأحكام لاحقاً لحساب الدقة الحية للنموذج، وهي المعيار الذي تُبنى عليه التوصية بإعادة التدريب.




الشكل 5:مخطط تتالي النظام لمراجعة تفاصيل التنبيه وتحديد الحكم

### 5.2.4- حالة استخدام: الاستعلام عن سُمعة عنوان IP

| | |
|---|---|
| اسم الحالة : الاستعلام عن | سمعة عنوان IP |
| الوصف (Description) | يستعلم النظام عن سجل السمعة الأمنية لعنوان IP معيّن من مصدر خارجي موثوق، ضمن سياق مراجعة تفاصيل تنبيه |
| الفاعلين (Actors) | Analyst/Admin بشكل غير مباشر |
| الشروط السابقة(Precondition) | تم فتح تفاصيل تنبيه يحتوي على عنوان IP عام (غير محلي) |
| الشروط اللاحقة (Postcondition) | تم إرفاق تقرير السمعة بتفاصيل التنبيه المعروضة |

جدول 5:حالة استخدام الاستعلام عن سمعة عنوان **ip**

**سير الأحداث (السيناريو الأساسي الناجح):**

| النظام |
|---|
| 1. تحقق النظام مما إذا كان عنوان الـ IP خاصاً (Private) أم عاماً. |
| 2. إن كان عاماً، يتحقق النظام من وجود نتيجة مخزَّنة مسبقاً لهذا العنوان خلال آخر 24 ساعة. |
| 3. إن لم توجد نتيجة حديثة، يستعلم النظام من خدمة AbuseIPDB الخارجية. |
| 4. يحفظ النظام النتيجة الجديدة مؤقتاً لإعادة استخدامها لاحقاً. |
| 5. يعيد النظام تقرير السمعة ليُعرَض ضمن تفاصيل التنبيه. |

المسارات البديلة:

A1: إن كان العنوان خاصاً (محلي)، يتجاوز النظام الاستعلام الخارجي ويعيد إشارة توضّح أن بيانات السمعة العامة غير متاحة لهذا النوع من العناوين.

A2: إن وُجدت نتيجة مخزَّنة حديثة (أقل من 24 ساعة)، يعيدها النظام مباشرة دون استعلام خارجي جديد.

مسارات الأخطاء:

E1: في حال فشل الاتصال بخدمة AbuseIPDB الخارجية، يعيد النظام إشارة خطأ ضمن التقرير دون إيقاف عرض بقية تفاصيل التنبيه.

### 6.2.4- حالة استخدام: حظر عنوان IP

| اسم الحالة : حظر عنوان IP | |
|---|---|
| الوصف (Description) | يقوم الأدمن بحظر عنوان IP يدوياً لمنعه من التفاعل مع النظام مستقبلاً |
| الفاعلين (Actors) | Analyst/Admin |
| الشروط السابقة(Precondition) | الأدمن مسجل الدخول ويحمل رمز جلسة صالحاً |
| الشروط اللاحقة (Postcondition) | أُضيف عنوان IP إلى قائمة العناوين المحظورة |

جدول 6 : حالة استخدام حظر عنوان **ip**

**سير الأحداث (السيناريو الأساسي الناجح):**

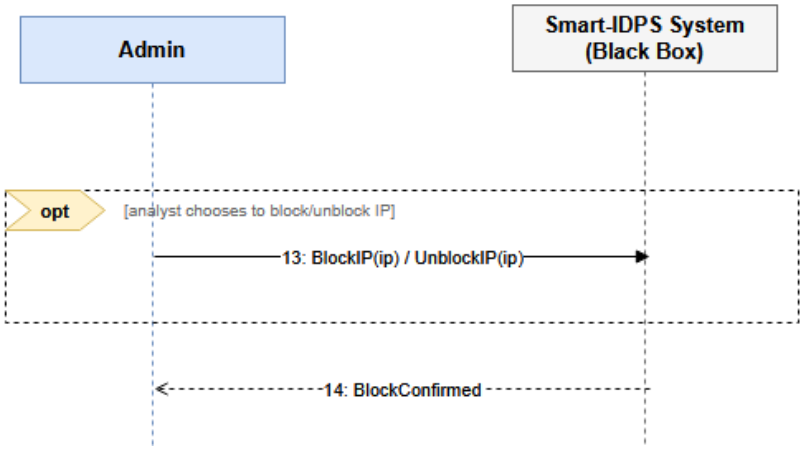
| الأدمن | النظام |
|---|---|
| 1. يطلب حظر عنوانIP . | |
| | 2.يتحقق النظام من صلاحية رمز الجلسة. |
| | 3.يُضيف النظام العنوان إلى قائمة العناوين المحظورة، ويؤكد نجاح العملية. |

المسارات البديلة:

A1: إن كان العنوان محظوراً مسبقاً، يتجاهل النظام الطلب دون إضافة سجل مكرَّر، مع تأكيد نجاح العملية.

مسارات الأخطاء:

E1: في حال كان رمز الجلسة غير صالح أو منتهياً، يرفض النظام الطلب ويطلب من الأدمن تسجيل الدخول من جديد.




الشكل**6** :مخطط تتالي النظام لحالة حظر عنوان **ip**

### 7.2.4- حالة استخدام: رفع الحظر عن عنوانIP

| اسم الحالة : رفع | الحظر عن عنوان IP |
|---|---|
| الوصف (Description) | يقوم الأدمن بإزالة عنوان IP من قائمة العناوين المحظورة |
| الفاعلين (Actors) | Analyst/Admin |
| الشروط السابقة(Precondition) | الأدمن مسجل الدخول، والعنوان موجود ضمن قائمة الحظر |

| | |
|---|---|
| الشروط اللاحقة (Postcondition) | أُزيل عنوان IP من قائمة العناوين المحظورة |

**سير الأحداث (السيناريو الأساسي الناجح):**

جدول 7:حالة استخدام رفع الحظر عن عنوان **ip**

| الأدمن | النظام |
|---|---|
| 1. يطلب رفع الحظر عن عنوان IP محدَّد. | |
| | 2.يتحقق النظام من صلاحية رمز الجلسة. |
| | 3. يحذف النظام العنوان من قائمة الحظر، ويؤكد نجاح العملية. |

المسارات البديلة: لا يوجد.

مسارات الأخطاء:

E1: في حال كان رمز الجلسة غير صالح، يرفض النظام الطلب.

### 8.2.4- حالة استخدام: عرض أداء النموذج

| | |
|---|---|
| اسم الحالة :عرض | أداء النموذج |
| الوصف (Description) | يستعرض الأدمن مؤشرات أداء النموذج الحية، المبنية على تراكم أحكام المراجعة البشرية، مع توصية صريحة بشأن الحاجة لإعادة التدريب |
| الفاعلين (Actors) | Analyst/Admin |
| الشروط السابقة(Precondition) | الأدمن مسجل الدخول. |
| الشروط اللاحقة (Postcondition) | تم عرض مؤشرات الأداء الحالية وتوصية إعادة التدريب |

جدول 8: حالة استخدام عرض أداء النموذج

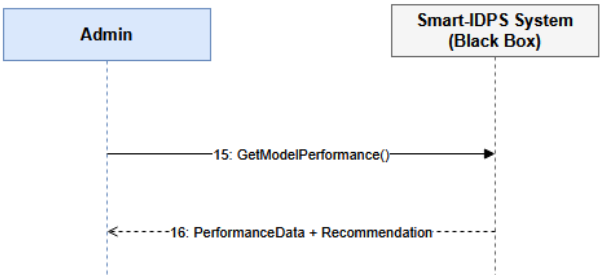
**سير الأحداث (السيناريو الأساسي الناجح):**

| الأدمن | النظام |
|---|---|
| 1. يفتح صفحة أداء النموذج. | |
| | 2. يحسب النظام الدقة الحية بناءً على نسبة الأحكام المؤكِّدة إلى إجمالي الأحكام المراجَعة، إجمالاً ولكل نوع هجوم على حدة. |
| | 3. يتحقق النظام مما إذا استُوفي أحد شرطي التوصية بإعادة التدريب (انخفاض الدقة عن الحد الأدنى، أو تجاوز عدد التنبيهات المعلَّقة الحد الأقصى). |
| | 4.يعرض النظام مؤشرات الأداء كاملة، مرفقة بالتوصية وأسبابها إن وُجدت. |

**المسارات البديلة:**

**A1: إن لم يتحقق أي من الشرطين، يعرض النظام رسالة تفيد بأن أداء النموذج ضمن النطاق المقبول حالياً.**

**مسارات الأخطاء: لا يوجد**




الشكل 7:مخطط تتالي النظام لحالة عرض أداء النموذج

### 9.2.4- حالة استخدام: تفعيل إعادة تدريب النموذج

| | |
|---|---|
| اسم الحالة :تفعيل | إعادة تدريب النموذج |
| الوصف (Description) | يُطلِق الأدمن عملية إعادة تدريب النموذج، بالاستناد إلى توصية يعرضها النظام مبنية على أداء النموذج الحي |

| | |
|---|---|
| الفاعلين (Actors) | Analyst/Admin |
| الشروط السابقة(Precondition) | الأدمن مسجل الدخول، ولا توجد عملية إعادة تدريب أخرى قيد التنفيذ حالياً |
| الشروط اللاحقة (Postcondition) | أُطلِقت مهمة إعادة تدريب في الخلفية، وتُحدَّث حالتها دورياً حتى الوصول لنتيجة نهائية |

الشكل 8: حالة استخدام تفعيل إعادة تدريب النموذج

**سير الأحداث**

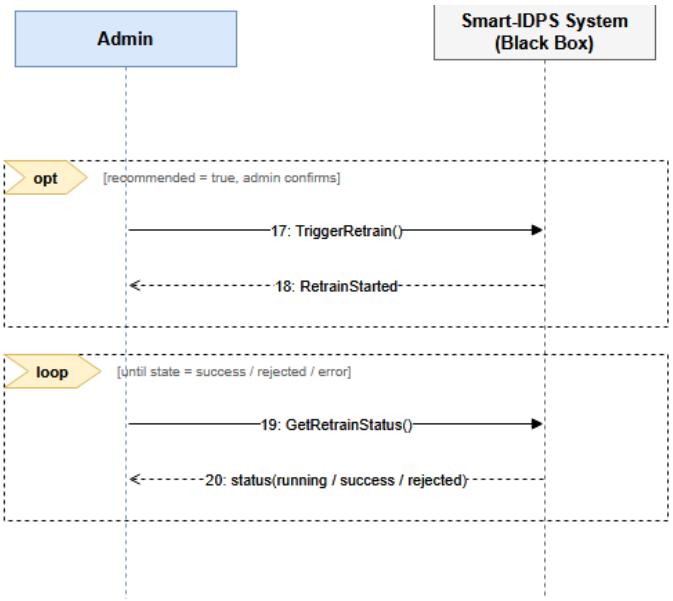
**(السيناريو الأساسي الناجح):**

| الأدمن | النظام |
|---|---|
| 1. يطّلع على توصية إعادة التدريب ضمن صفحة أداء النموذج. | |
| 2. يقرر تفعيل إعادة التدريب. | |
| | 3 .يتحقق النظام من عدم وجود عملية إعادة تدريب أخرى قيد التنفيذ. |
| | 4 .يُطلِق النظام مهمة إعادة التدريب في الخلفية، ويعيد استجابة فورية ببدء العملية. |
| | 5. تستطلع الواجهة حالة التقدّم دورياً حتى تصل النتيجة النهائية (ترقية أو رفض). |

المسارات البديلة: لا يوجد.

مسارات الأخطاء:

E1: في حال وجود عملية إعادة تدريب أخرى قيد التنفيذ بالفعل (الخطوة 3)، يرفض النظام الطلب برسالة توضيحية.




الشكل 9: حالة استخدام: تفعيل إعادة تدريب النموذج

### 10.2.4- حالة استخدام: تصدير التنبيهات المُراجَعة

| | |
|---|---|
| اسم الحالة : | تصدير التنبيهات المراجَعة |
| الوصف (Description) | يقوم الأدمن بتصدير جميع التنبيهات التي صدر بشأنها حكم بشري (مؤكَّد أو كاذب) إلى ملف CSV لاستخدامها لاحقاً في إعادة التدريب أو التحليل الخارجي |
| الفاعلين (Actors) | Analyst/Admin |
| الشروط السابقة(Precondition) | الأدمن مسجل الدخول |
| الشروط اللاحقة (Postcondition) | تم تنزيل ملف يحتوي على جميع التنبيهات المراجَعة |

جدول 9: حالة استخدام تصدير التنبيهات المراجَعة

**سير الأحداث (السيناريو الأساسي الناجح):**

| الأدمن | النظام |
|---|---|
| 1. يطلب تصدير التنبيهات المراجَعة. | |
| | 2. يستعلم النظام عن جميع التنبيهات التي تحمل حكماً بشرياً نهائياً. |

| | |
|---|---|
| | 3.يُنشئ النظام ملف CSV يحتوي على هذه التنبيهات، ويُتيحه للتنزيل. |

المسارات البديلة: لا يوجد.

مسارات الأخطاء: لا يوجد.

# الفصل الخامس

# تصميم النظام

*نعرض في هذا الفصل القرارات التصميمية التي تم الاعتماد عليها لبناء النظام.*

## 1.5- مقدمة

تُبرز مرحلة تصميم النظام القدرة على تحويل الأهداف النظرية والمتطلبات التي تم تحديدها سابقاً إلى قرارات هندسية واضحة تحدد بنية المكوّنات وطريقة تفاعلها فيما بينها، بما يضمن تحقيق المتطلبات الوظيفية وغير الوظيفية على حد سواء. وبما أن نظام يستهدف مراقبة حركة الشبكة في بيئة سحابية تتصف بضخامة حجم البيانات المتدفقة وتعدد مصادرها، وبضرورة الكشف عن الهجمات ضمن نافذة زمنية قريبة من الزمن الحقيقي، فقد كان من الضروري اعتماد بنية موزّعة (Distributed Architecture) بدلاً من الاعتماد على معالجة مركزية أحادية العقدة.

يستند هذا القرار إلى أساس أكاديمي واضح: فالأنظمة الأمنية التقليدية التي تعتمد على خادم واحد لتحليل السجلات تصطدم بحاجز فيزيائي في قدرتها على المعالجة عند تزايد حجم الحركة الشبكية، إذ يتحوّل الخادم إلى نقطة اختناق (Bottleneck) ونقطة فشل وحيدة (Single Point of Failure) في آن معاً — وهو تناقض جوهري مع طبيعة النظام الأمني الذي يُفترض أن يبقى متاحاً ومستقراً في كل الأوقات، وتحديداً أثناء وقوع الهجوم. لهذا السبب اعتمد المشروع على Apache Hadoop كطبقة تخزين موزّع (HDFS)، توفّر تحمّلاً للأعطال عبر آلية النسخ المتماثل بين العقد، وتسمح بأرشفة كميات متنامية من سجلات الحركة الشبكية دون أن يشكّل نمو البيانات عبئاً على أداء النظام. وبالتوازي، اعتمد المشروع على Apache Spark كمحرك معالجة موزّع يعمل في الذاكرة (In-Memory Processing)، وهو ما يمنحه تفوقاً واضحاً على نماذج المعالجة الدفعية التقليدية القائمة على القرص (مثل MapReduce)، ويجعله الخيار الأنسب لتحليل الحركة الشبكية وتطبيق نموذج الذكاء الاصطناعي عليها في زمن قريب من اللحظي، بدلاً من انتظار تجميع البيانات ثم تحليلها لاحقاً.

انطلاقاً من ذلك، اعتمد تصميم النظام على منهج علمي مبني على مقارنات دقيقة لاختيار الأدوات والتقنيات الأنسب لكل مرحلة من مراحل معالجة البيانات، بدءاً من استيعاب حركة الشبكة القادمة من عدة مصادر بشكل متزامن عبر Apache Kafka، مروراً بتخزينها بطريقة موزّعة ضمن HDFS، ووصولاً إلى تحليل هذه البيانات عبر Spark باستخدام نموذج ذكاء اصطناعي هرمي ثنائي المرحلة، ثم عرض النتائج للمشرف واتخاذ إجراء الاستجابة المناسب.

يستعرض هذا الفصل مكوّنات النظام ودور كل مكوّن، ثم يناقش التحديات المعمارية التي واجهت عملية التصميم والحلول المعتمدة لمعالجتها، ويوضح تدفق البيانات داخل النظام من لحظة توليد الحركة الشبكية وحتى صدور التنبيه الأمني، وصولاً إلى الأسس التي تحقق قابلية التوسّع والاعتمادية في النظام ككل.

## 2.5- النمط المعماري المعتمد

قبل الخوض في تفاصيل مكوّنات النظام، من الضروري تحديد النمط المعماري الذي بُني عليه ، إذ إن اختيار هذا النمط هو القرار الجذري الذي انبثقت عنه كل القرارات التقنية اللاحقة.

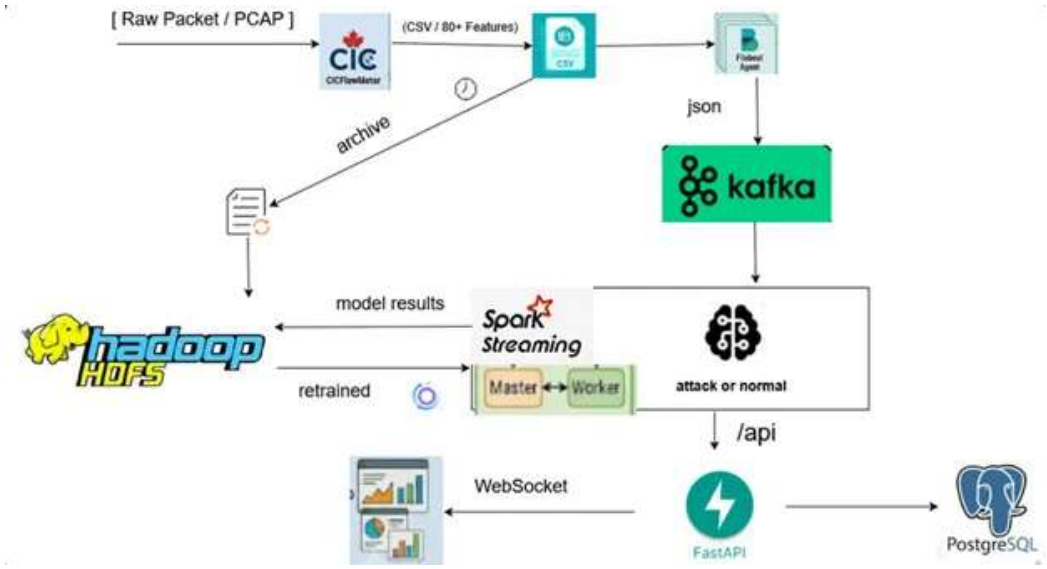
يواجه أي نظام أمني موزّع تناقضاً جوهرياً بين متطلبين يبدوان متعارضين في ظاهرهما: الحاجة إلى الكشف اللحظي عن الهجمات فور وقوعها، والحاجة في الوقت ذاته إلى الاحتفاظ بأرشيف تاريخي كامل للحركة الشبكية بغرض التحليل الجنائي الرقمي (Forensics) وإعادة تدريب النموذج على بيانات واقعية. الاعتماد على مسار معالجة واحد فقط لتحقيق الهدفين معاً يُنتج حلاً وسطاً ضعيفاً: فإما أن يُضحّى بسرعة الكشف لصالح اكتمال الأرشفة، أو يُضحّى باكتمال الأرشفة لصالح السرعة.

لهذا السبب اعتمد تصميم النظام على مبدأ الفصل بين مسار المعالجة اللحظية ومسار المعالجة التاريخية (Speed Path vs. Batch Path)، وهو مبدأ معماري راسخ في أنظمة معالجة البيانات الضخمة، يقوم على تغذية مسارين متوازيين من مصدر بيانات واحد: مسار سريع يخدم القرار الفوري (هل هذا التدفق هجوم أم لا؟)، ومسار أبطأ يخدم الأرشفة طويلة الأمد وإعادة البناء. هذا الفصل يسمح لكل مسار بأن يُحسَّن (Optimize) بمعزل عن الآخر: فمسار الكشف اللحظي يُبنى على أدوات تعطي الأولوية لزمن الاستجابة (Kafka وSpark Streaming)، بينما يُبنى مسار الأرشفة على أدوات تعطي الأولوية لسعة التخزين وتحمّل الأعطال (Hadoop HDFS)، دون أن يفرض أحد المسارَيْن قيوده على أداء الآخر.

بالإضافة إلى هذا الفصل الأفقي بين المسارين، اعتمد النظام أيضاً على تقسيم عمودي للمعالجة عبر بنية متعددة الطبقات ( Layered Pipeline Architecture)، بحيث تمر البيانات عبر سلسلة من الطبقات المتخصصة والمستقلة — استخلاص الخصائص، الاستيعاب والبث، التحليل الذكي، التخزين، ثم العرض — بحيث تتصل كل طبقة بالطبقة المجاورة لها فقط عبر واجهة أو وسيط محدد (Kafka بين الاستيعاب والتحليل، مثلاً)، دون أن تعرف أي طبقة تفاصيل التنفيذ الداخلي للطبقات الأخرى. هذا الفصل يحقق ميزتين هندسيتين أساسيتين: الأولى هي إمكانية استبدال أو توسيع أي طبقة بمعزل عن غيرها (كإضافة عقد Spark Worker جديدة دون التأثير على منطق الاستيعاب)، والثانية هي منع أي طبقة من التحوّل إلى نقطة اختناق (Bottleneck) تُبطئ النظام ككل، بما يتوافق مع متطلب قابلية التوسّع الذي تم تحديده في دفتر الشروط.

## 3.5- مكونات النظام

بعد استعراض النمط المعماري العام والمنطق الذي بُنيت عليه البنية الموزّعة للنظام، يتناول هذا القسم كل مكوّن من مكوّنات النظام بالتفصيل، موضّحاً دوره الوظيفي داخل خط سير البيانات (Data Pipeline) والمبرر الهندسي لاختياره تحديداً دون بدائله.




الشكل 10:مكونات النظام

يعرض الشكل (10) هذه المكوّنات مرتّبة وفق تسلسل معالجة البيانات، من لحظة التقاط الحركة الشبكية الخام وحتى صدور التنبيه وعرضه للمشرف ،مبررين أسباب اختيار هذه المكونات دوناً عن غيرها.

- **CICFlowMeter — استخلاص الميزات**

يمثّل CICFlowMeter نقطة الدخول إلى النظام، إذ يستقبل حركة الشبكة الخام بصيغة حزم (Raw Packets / PCAP) ويحوّلها إلى تدفقات شبكية (Network Flows) موصوفة بأكثر من ثمانين خاصية إحصائية، مثل مدة التدفق، عدد الحزم في كل اتجاه، معدل نقل البايتات، وأعلام TCP. اختيار هذه الأداة تحديداً — بدلاً من كتابة منطق استخلاص خصائص مخصَّص — يستند إلى اعتبار منهجي دقيق: فقد استُخدمت النسخة ذاتها من CICFlowMeter في بناء مجموعة البيانات CSE-CIC-IDS2018 التي دُرِّب عليها النموذج، مما يضمن تطابقاً كاملاً بين خصائص الحركة الحية وخصائص بيانات التدريب من حيث العدد والترتيب والوحدة، وهو شرط أساسي لصحة عملية التصنيف؛ إذ إن أي اختلاف في طريقة استخلاص الخصائص بين بيئة التدريب وبيئة التشغيل الفعلي (Training–Serving Skew) يؤدي إلى تدهور دقة النموذج بغض النظر عن جودته.

- **Filebeat — وكيل الاستيعاب**

بعد أن تُختزَن التدفقات الناتجة بصيغة CSV، يتولى وكيل Filebeat مراقبتها وقراءتها فور تحديثها، ثم تحويلها إلى صيغة JSON ونقلها إلى وسيط الرسائل Kafka. يُعفي هذا الوكيل النظام من الحاجة إلى كتابة منطق مراقبة ملفات مخصَّص (File Watching)، ويوفر آلية موثوقة لضمان عدم فقدان أي سطر جديد يُضاف إلى ملف CSV، فضلاً عن قدرته على العمل كعملية خفيفة الموارد مقارنة ببدائل أثقل.

- **Apache Kafka**

يستقبل Kafka الأحداث القادمة من Filebeat بصيغة JSON ويعمل كطبقة عازلة (Buffering Layer) بين مصدر البيانات ومستهلكها. هذا الفصل يحقق ميزتين: الأولى استيعاب أي تفاوت مؤقت بين معدّل توليد الأحداث ومعدّل معالجتها، بحيث لا تُفقَد أي بيانات عند حدوث ارتفاع مفاجئ في حجم الحركة الشبكية — وهي بالضبط الحالة التي تحدث أثناء وقوع هجوم مثل DDoS. والثانية إمكانية إضافة أكثر من مستهلك (Consumer) لاحقاً يقرأ من الموضوع (Topic) ذاته دون أي تعديل في مصدر البيانات، بما يخدم متطلب قابلية التوسّع.

- **Apache Spark Streaming — محرك التحليل الذكي**

يمثّل Spark Streaming، العامل ضمن بنية موزّعة من نوع Master–Worker، جوهر النظام ومسؤول عن قراءة الأحداث من Kafka فور وصولها وتمريرها على النموذج الهرمي ثنائي المرحلة لتصنيف كل تدفق (attack or normal)، ثم تحديد نوع الهجوم عند الاقتضاء. اعتماد Spark في هذه الطبقة تحديداً مبرَّر باعتماده على المعالجة داخل الذاكرة (In-Memory Computation)، وهو ما يمنحه أداءً أعلى من نماذج المعالجة الدفعية التقليدية القائمة على القرص، إضافة إلى أن بنيته الموزّعة تتيح توزيع عبء التصنيف على عدة عقد Worker بالتوازي، بحيث يمكن للنظام أن يتوسّع أفقياً بإضافة عقد جديدة دون أي تعديل في منطق التحليل نفسه.

يرتبط Spark بمخزن HDFS بعلاقة تخدم دورة حياة النموذج بالكامل:

- مسار الإخراج (Model Results): بعد كل عملية تصنيف، تُحفَظ نتائج النموذج في HDFS، بما يُنشئ أرشيفاً تراكمياً من قرارات النظام يُستخدم لاحقاً في التحليل والمراجعة.
- مسار إعادة التدريب (Retrained Model): يتيح هذا المسار تحميل نسخة مُحدَّثة من النموذج بعد إعادة تدريبه على بيانات واقعية جُمعت من بيئة التشغيل الفعلي، ليحل محل النموذج الحالي داخل Spark دون إيقاف النظام. هذا المسار يُجسّد عملياً استجابة النظام لأحد أبرز محدوديات أي نموذج مُدرَّب على مجموعة بيانات ثابتة، وهي احتمال تراجع أدائه أمام أنماط هجوم جديدة لم يرَها أثناء التدريب.

- **Hadoop HDFS — طبقة الأرشفة الموزّعة**

يؤدي HDFS في هذا النظام دورين متكاملين. الدور الأول هو الأرشفة الدورية (Periodic Archival) للتدفقات الخام الناتجة عن CICFlowMeter، بما يوفر سجلاً تاريخياً كاملاً للحركة الشبكية يمكن الرجوع إليه لأغراض التحليل الجنائي الرقمي (Forensics) أو إعادة بناء سياق حادثة أمنية سابقة. أما الدور الثاني فهو تخزين مخرجات النموذج ونماذجه المعاد تدريبها، كما ورد سابقاً. اختيار HDFS تحديداً لهذين الغرضين مبرَّر بقدرته على توزيع البيانات وتكرارها (Replication) عبر عدة عقد تخزين، بما يضمن استمرارية توفر السجلات حتى عند تعطّل إحدى هذه العقد — وهي خاصية جوهرية في نظام أمني لا يجوز أن يفقد أي دليل رقمي مرتبط بحادثة أمنية.

- **FastAPI — طبقة الخدمة الخلفية back end**

يشكّل FastAPI نقطة الوصل الوحيدة بين طبقة التحليل الذكي وطبقتَي التخزين التشغيلي والعرض، ولا يُسمح لأي مكوّن آخر بالاتصال المباشر بقاعدة البيانات أو بالواجهة الأمامية. عند اكتشاف Spark لهجوم، يُرسِل طلب HTTP إلى FastAPI عبر واجهة برمجية مخصَّصة (/api/alerts)، حيث يتولى FastAPI معالجة منطق الأعمال الكامل المرتبط بالتنبيه (كتجميع التنبيهات المتكررة ضمن نافذة زمنية محددة لتفادي إغراق المشرف بتنبيهات مكرّرة عن الهجوم ذاته)، قبل أن يكتب السجل في قاعدة البيانات.

يستند هذا التصميم إلى مبدأ الفصل بين منطق التحليل ومنطق التخزين والأعمال، بحيث تبقى مسؤولية Spark محصورة في التصنيف فقط، بينما تُدار كل القرارات المتعلقة بكيفية معالجة نتيجة هذا التصنيف (تخزين، إشعار، صلاحيات وصول) في طبقة منفصلة قابلة للتعديل دون المساس بمنطق النموذج. كما يحقق هذا الفصل حداً أمنياً أساسياً، إذ يمنع كشف بيانات الاتصال بقاعدة البيانات لأي طبقة خارج الخدمة الخلفية— وهو اعتبار جوهري في نظام أمني بطبيعته.

- **PostgreSQL — قاعدة البيانات التشغيلية**

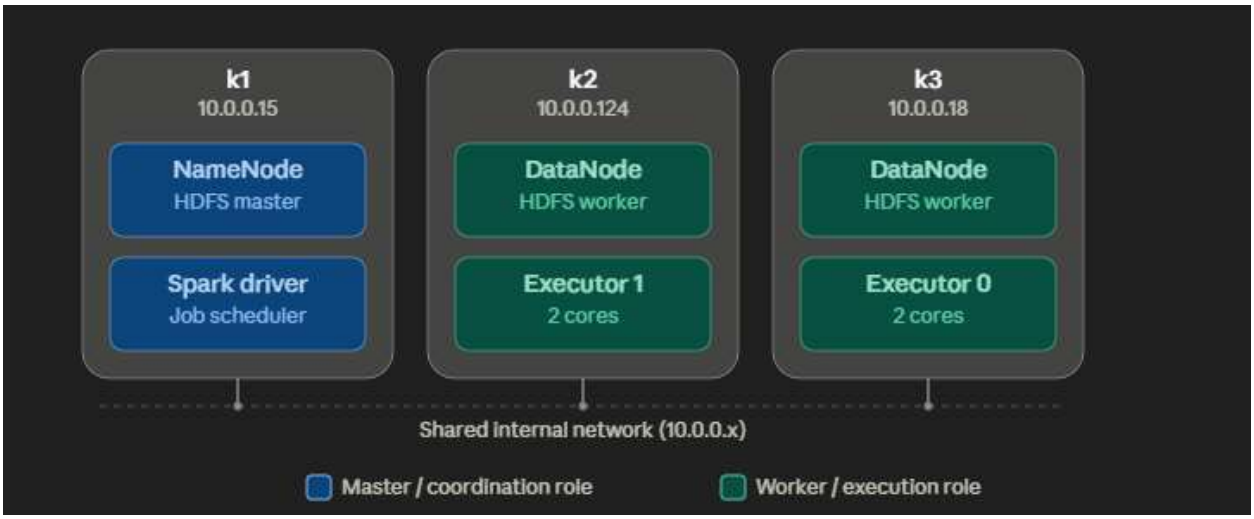
تُستخدم PostgreSQL لتخزين التنبيهات (Alerts) بشكل منظَّم يسمح باستعلامات دقيقة وسريعة، كتصفية التنبيهات حسب النوع أو مستوى الخطورة أو الفترة الزمنية. اختيار قاعدة بيانات علائقية لهذا الغرض تحديداً — بدلاً من الاعتماد على HDFS نفسه — مبرَّر باختلاف طبيعة الوصول إلى البيانات في كل حالة: فبينما تخدم HDFS الأرشفة الضخمة غير المهيكلة، تخدم PostgreSQL الاستعلامات المتكررة السريعة على بيانات تشغيلية محدودة الحجم نسبياً، وهي نمط وصول تتفوق فيه قواعد البيانات العلائقية بوضوح.

- **لوحة التحكم (Dashboard)**

تمثّل لوحة التحكم الواجهة النهائية التي يتفاعل معها المشرف، وترتبط بـ FastAPI عبر قناتَي اتصال منفصلتَي الغرض: طلب HTTP عادي (Request–Response) يُستخدم عند فتح الصفحة لأول مرة لجلب السجل التاريخي للتنبيهات المخزَّنة في PostgreSQL، واتصال WebSocket دائم ومفتوح يُستخدم لدفع (Push) أي تنبيه جديد فور صدوره من Spark، دون أن يحتاج المستخدم إلى تحديث الصفحة يدوياً. هذا التمييز بين نمطَي الاتصال يعكس طبيعة كل حاجة: الأول ملائم لجلب بيانات تاريخية ثابتة عند الطلب، والثاني ضروري لتحقيق متطلب الكشف والعرض في زمن قريب من اللحظي، وهو الهدف الجوهري للنظام بأكمله.

## 3.5- معمارية العنقود الموزع: نمط السيادة والتنفيذ (Master-Worker Cluster Architecture)

يعتمد كل من Apache Hadoop وApache Spark في هذا المشروع على نمط معماري موحّد Master-Worker Architecture، وهو نمط تنظيمي يقوم على تقسيم مسؤوليات العنقود إلى فئتين وظيفيتين متمايزتين تماماً: عقدة سيادة واحدة تتولى التنسيق المركزي دون المشاركة المباشرة في العبء الحسابي، وعقد تنفيذ متعددة تتولى المعالجة الفعلية وتنفيذ المهام الموزَّعة عليها.

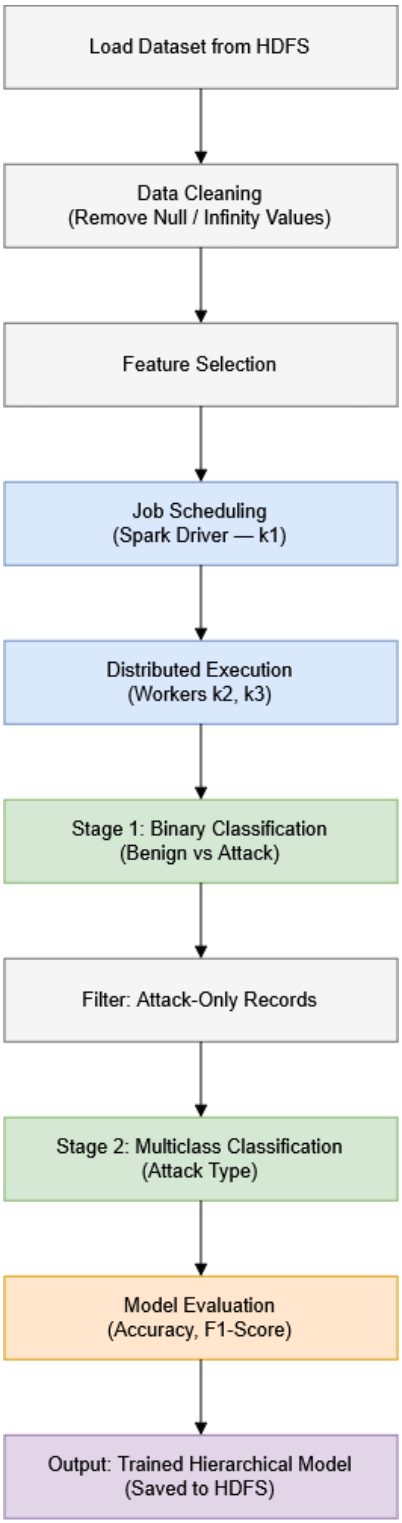



تتجسّد عقدة السيادة (Master Node) في هذا المشروع بالعقدة k1 (10.0.0.15)، التي تحمل دوراً مزدوجاً محورياً: فهي NameNode بالنسبة لنظام Hadoop، مسؤولة عن إدارة مساحة الأسماء (Namespace) لنظام الملفات الموزَّع وتتبّع مواقع كتل البيانات المخزَّنة فعلياً على عقد التنفيذ، وهي في الوقت ذاته Spark Driver، مسؤولة عن جدولة المهام الحسابية وتوزيعها على العقد المنفِّذة وتجميع النتائج الجزئية العائدة منها. ومن الجدير التأكيد أن هذه العقدة، رغم مركزيتها التنظيمية، لا تُخصِّص لها أي عبء تخزيني أو حسابي مباشر — فهي "تُنسِّق" العمل ولا "تُنفِّذه".

في المقابل، تتجسّد عقد التنفيذ (Worker Nodes) بالعقدتين k2 (10.0.0.124) وk3 (10.0.0.18)، اللتين تحملان بدورهما دوراً مزدوجاً متماثلاً: فكل منهما DataNode بالنسبة لـ Hadoop، مسؤولة عن التخزين الفعلي لكتل البيانات المجزَّأة (Data Blocks)، وفي الوقت ذاته Spark Executor، مسؤولة عن التنفيذ الفعلي للحسابات الموزَّعة بواقع نواتين معالجة (2 Cores) لكل عقدة، أي بقدرة حوسبية إجمالية فعلية تبلغ 4 أنوية عبر العنقود بأكمله.

وتُبرز هذه البنية قراراً تصميمياً مقصوداً وليس تنظيماً عشوائياً: فتطابق الأدوار عبر النظامين — بحيث تحمل العقدة نفسها دور السيادة أو التنفيذ في كليهما معاً وليس في أحدهما فقط — يُقلِّل من تعقيد إدارة الشبكة الداخلية للعنقود، إذ تصبح العقدة التي تُنسِّق موقع البيانات (NameNode) هي ذاتها التي تُنسِّق معالجتها (Spark Driver)، مما يُختصر مسارات الاتصال الداخلي ويُخفِّف من الحمل الشبكي الناتج عن التنسيق المتقاطع بين "منسّقين" مختلفين لنظامين منفصلين.

## 4.5- تصميم نموذج الذكاء الصنعي




الشكل**11** :تصميم تدريب نموذج الذكاء الصنعي

يوضّح الشكل (11) خط الأنابيب (Pipeline) المعتمَد لتصميم وتدريب نموذج الذكاء الاصطناعي داخل النظام، بدءاً من مصدر البيانات وصولاً إلى النموذج النهائي الجاهز للاستخدام.

تبدأ العملية بتحميل مجموعة البيانات من HDFS، حيث تُقرَأ البيانات الخام مباشرة من نظام الملفات الموزَّع الذي يحفظ السجلات الشبكية بشكل دائم. تلي ذلك مرحلة تنظيف البيانات، وتشمل إزالة القيم اللانهائية والمفقودة والتسميات التالفة، تمهيداً لمرحلة اختيار الميزات التي تُحدِّد الخصائص الشبكية الأنسب لتدريب النموذج واستبعاد ما لا يخدم عملية التصنيف أو قد يتسبب في الإفراط في التخصيص (Overfitting).

بعد تجهيز البيانات، تُسلَّم مهمة التدريب لعقدة k1 التي تتولى دور Spark Driver المسؤول عن جدولة المهمة، لتُوزَّع بعدها عملية التنفيذ الفعلي على عقدتي k2 وk3 بشكل متوازٍ، وفق نمط Master-Worker المعتمد في العنقود. وتسير عملية التصنيف على مرحلتين متعاقبتين: تُنفَّذ المرحلة الأولى تصنيفاً ثنائياً يميّز بين الحركة الطبيعية والهجوم، ثم تُصفَّى السجلات المصنَّفة كهجوم فقط لتُمرَّر إلى المرحلة الثانية التي تُحدِّد نوع الهجوم تحديداً من بين الفئات الهجومية المختلفة.

وأخيراً، يخضع النموذج الناتج لمرحلة تقييم باستخدام مقاييس الدقة وF1-Score للتحقق من جودة أدائه، قبل أن يُحفَظ النموذج النهائي بمرحلتيه على HDFS، ليكون جاهزاً للاستخدام لاحقاً في عملية الاستدلال اللحظي عند تشغيل النظام.

## 5.5- الأنماط التصميمية المعتمدة

اعتمد تصميم النظام على مجموعة من الأنماط المعمارية التي أثبتت فعاليتها في بناء أنظمة معالجة البيانات الموزَّعة والمعتمِدة على الوقت الفعلي. فيما يلي استعراض لأبرز هذه الأنماط كما طُبِّقت داخل المشروع:

- **النمط القائم على الأحداث (Event-Driven Architecture)**

اعتُمد النمط القائم على الأحداث كأسلوب أساسي لتبادل البيانات بين مكوّنات النظام، بدلاً من الاعتماد على استدعاءات متزامنة ومباشرة (Synchronous Calls) بين طبقة التقاط السجلات وطبقة معالجتها. يعتمد هذا النمط على مبدأ الناشر/المشترك (Publisher/Subscriber)، حيث تقوم Filebeat (بصفته الناشر) بنشر السجلات الشبكية كأحداث إلى وسيط الرسائل Kafka، دون أن تعرف أو تنتظر من سيقوم بمعالجة هذه الأحداث لاحقاً. وفي المقابل، يقوم Spark Streaming (بصفته المشترك) بالاستماع إلى هذه الأحداث وتنفيذ عملية التصنيف عليها فور وصولها، مما يحقق تواصلاً غير مباشر (Indirect Communication) بين طبقتي جمع البيانات ومعالجتها.

فحين يلتقط سجلاً شبكياً جديداً، يُنشَر هذا السجل كحدث إلى Kafka دون أي معرفة مسبقة بمن سيستهلكه؛ ليقوم Spark Streaming باستهلاكه وتحليله عبر النموذج الهرمي، بينما تستهلك في الوقت نفسه خدمة الأرشفة (HDFS) الحدث ذاته لتخزينه بشكل دائم، دون أن يتطلب ذلك أي تنسيق مباشر بين الطرفين.

يحقق هذا النمط عدة فوائد جوهرية للنظام:

- فصل منطقي وزمني بين المكوّنات، بحيث لا يُشترط أن تكون خدمة المعالجة (Spark) متاحة لحظة وصول السجل، إذ يبقى محفوظاً في Kafka إلى حين استعداد المستهلك لمعالجته.
- تقليل الترابط (Loose Coupling)، مما يسمح بتعديل أو استبدال أي مكوّن — كتغيير خوارزمية التصنيف داخل Spark أو إضافة مسار معالجة جديد — دون أي تأثير مباشر على مصدر البيانات أو بقية المكوّنات المستهلِكة.

- مراعاة اختلاف السرعات بين المكوّنات؛ فتوليد السجلات قد يكون متواصلاً وسريعاً، بينما تحتاج عملية التصنيف الهرمي بمرحلتيها وقتاً أطول نسبياً لمعالجة كل دفعة، وهنا يعمل Kafka كطبقة عازلة (Buffer) تسمح لكل مكوّن بالعمل وفق وتيرته الخاصة دون أن يُبطئ أحدهما الآخر.
- تحسين قابلية التوسع (Scalability)، إذ يمكن إضافة مستهلكين جدد للأحداث ذاتها — كخدمة تنبيه إضافية أو نظام أرشفة موازٍ — دون أي تعديل على السيرفرات الافتراضية أو منطق نشرها للسجلات.

- **نمط الفصل بين الطبقات (Layered Architecture)**

اعتُمد نمط الفصل بين الطبقات لتنظيم مكوّنات النظام ضمن تسلسل وظيفي واضح، بحيث تختص كل طبقة بمسؤولية محدَّدة دون تداخل مباشر مع مسؤوليات الطبقات الأخرى. يتكوّن النظام وفق هذا النمط من خمس طبقات متعاقبة:

طبقة جمع البيانات، وتضم CIC FLOW METER وFilebeat، وتختص حصراً بالتقاطها السجلات الشبكية من مصدرها.

طبقة النقل، ويمثّلها Kafka، وتختص بنقل الأحداث بين طبقة الجمع وطبقات المعالجة والتخزين دون معرفة بمحتوى البيانات أو الغاية النهائية منها.

طبقة التخزين، وتضم HDFS للتخزين الدائم الموزَّع، وتختص بحفظ السجلات الخام لأغراض الأرشفة وإعادة التدريب المستقبلي.

طبقة المعالجة والذكاء الاصطناعي، ويمثّلها Spark بعنقوده الموزَّع، وتختص بتطبيق النموذج الهرمي على السجلات الواردة واتخاذ قرار التصنيف.

طبقة العرض والتفاعل، وتضم FastAPI ولوحة التحكم، وتختص بعرض النتائج على المحلل الأمني واستقبال قراراته (كالحظر اليدوي أو التحقق من التنبيهات).

هذا الفصل الصارم بين الطبقات يحصر أثر أي تعديل ضمن حدود الطبقة المعنيّة به دون أن يمتد إلى بقية النظام؛ فتعديل آلية عرض التنبيهات في لوحة التحكم لا يستدعي أي تغيير في منطق الكشف داخل Spark، كما أن استبدال خوارزمية التصنيف المعتمدة في طبقة المعالجة لا يفرض أي تعديل على طبقتي الجمع أو النقل. ويحقق هذا النمط، إلى جانب وضوح المسؤوليات، تسهيلاً كبيراً في اختبار كل طبقة بمعزل عن غيرها، وقابلية أعلى لتوسيع أو استبدال أي طبقة مستقبلاً دون إعادة بناء النظام بأكمله

- **نمط تكرار النموذج (Model Replication Pattern)**

خلافاً لتوزيع النموذج نفسه بين العقد (Model Parallelism)، اعتُمد نمط تكرار النموذج، حيث تُنشَر نسخة كاملة ومطابقة من النموذج النهائي المدرَّب على كل من العقدتين k2 وk3 بشكل مستقل وذاتي الاكتفاء. يتيح هذا النمط لكل عقدة معالجة تدفقات السجلات الواردة إليها محلياً وبالتوازي التام مع العقدة الأخرى، دون الحاجة لأي تنسيق مركزي لحظة الاستدلال، مما يرفع الإنتاجية الكلية للنظام (Throughput) بدلاً من الاعتماد على نسخة مركزية واحدة قد تتحوّل إلى نقطة اختناق.

- **نمط الحفاظ على الحالة (Checkpointing Pattern)**

لضمان استمرارية معالجة تدفق البيانات دون فقدان أو تكرار عند حدوث انقطاع أو إعادة تشغيل، اعتمد Spark Streaming آلية تسجيل نقاط الحفظ (Checkpointing)، حيث تُحفَظ حالة المعالجة الحالية (كموضع آخر سجل مُعالَج من Kafka) بشكل دوري على القرص. وعند إعادة تشغيل خدمة المعالجة لأي سبب، يستأنف النظام العمل من آخر نقطة محفوظة بدلاً من البدء من جديد أو فقدان السجلات التي وصلت أثناء التوقف، وهو ما يضمن اتساق البيانات (Data Consistency) واستمرارية عملية الكشف عن الهجمات في بيئة تشغيل مستمرة على مدار الساعة.

## 6.5- تصميم التخزين حسب طبيعة البيانات

- **HDFS للسجلات الخام والأرشيف التاريخي**

تُستخدَم HDFS لتخزين السجلات الشبكية الخام بشكل دائم، وهي بيانات ضخمة الحجم تصل إلى ملايين السجلات يومياً، نادرة التعديل بعد كتابتها (Write-Once)، وتُقرَأ لاحقاً بشكل دفعي (Batch) لا فوري، تحديداً عند إعادة تدريب النموذج على بيانات تاريخية متراكمة. وقد اختيرت HDFS تحديداً لهذا الغرض لثلاثة أسباب:

- التخزين الموزَّع المتسامح مع الأعطال: تُقسِّم HDFS الملفات إلى كتل (Blocks) موزَّعة عبر عقد متعددة، بحيث لا يؤدي تعطّل عقدة واحدة إلى فقدان البيانات، وهي خاصية ضرورية لأرشيف يُفترض أن يبقى متاحاً على المدى الطويل.

- التكامل المباشر مع Spark: بما أن عملية إعادة التدريب تُنفَّذ عبر Spark، فإن قراءة البيانات من HDFS مباشرة تستفيد من محلية البيانات (Data Locality)، حيث تُنفَّذ المعالجة على نفس العقد التي تخزن البيانات فعلياً، دون نقلها عبر الشبكة أولاً.

- الملاءمة لحجم البيانات وطبيعتها: بيانات السجلات هنا أقرب لملفات مجمَّعة كبيرة الحجم من كونها جداول علائقية دقيقة تحتاج استعلاماً فورياً، وHDFS مصمَّم أصلاً لتخزين هذا النوع من الملفات بكفاءة.

- PostgreSQL للتنبيهات وبيانات التشغيل اللحظي

- خلافاً للسجلات الخام، تحتاج بيانات التشغيل اللحظي — التنبيهات (Alerts)، قرارات المحلل (Verdict)، عناوين IP المحظورة، وذاكرة سُمعة العناوين (IP Reputation Cache) — إلى نمط وصول مختلف تماماً: قراءات وكتابات متكررة وسريعة، استعلامات دقيقة ومُفلترة، وعلاقات منطقية واضحة بين البيانات. وقد رُجِّحت PostgreSQL لهذا الدور للأسباب التالية، التي تعكسها بنية الكود فعلياً:

- دعم الاستعلامات العلائقية الدقيقة والمركّبة: يعتمد النظام على استعلامات تجمع شروطاً متعددة في آنٍ واحد، كالفلترة حسب درجة الخطورة ونوع الهجوم وقرار المحلل معاً في نقطة استرجاع التنبيهات، أو تجميعات إحصائية لحساب عدد التنبيهات حسب النوع وتحديد أكثر عناوين IP نشاطاً، وهي عمليات تُعَدّ من صميم قوة قواعد البيانات العلائقية.

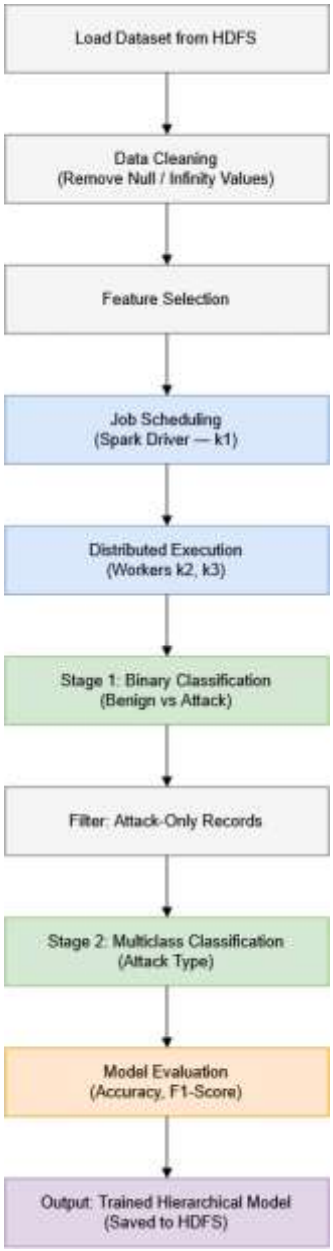
- دعم بيانات متغيّرة الشكل دون فقدان البنية العلائقية: استُخدم نوع عمود مخصص للبيانات شبه المهيكَلة (JSONB) لتخزين تفاصيل الخصائص الشبكية الخام المرافقة لكل تنبيه (sample_features) ضمن الجدول الرئيسي نفسه، مما منح النظام مرونة تخزين بيانات متغيّرة البنية دون التضحية بالتنظيم العلائقي لبقية الحقول.

- ضمانات التكامل والتزامن (ACID): تعتمد عمليات كتحديث التنبيه المتكرر ضمن نافذة زمنية محددة (منطق التجميع المطبَّق عند وصول تنبيهات متتالية لنفس نوع الهجوم من نفس المصدر) على قراءة دقيقة ثم تحديث لسجل واحد بعينه، وهي عملية تحتاج ضمانات تكامل صارمة توفرها قاعدة البيانات العلائقية بشكل أصيل، لضمان عدم تضارب التحديثات المتزامنة على نفس السجل.

كفاءة استهلاك الموارد: فُضِّلت PostgreSQL لاستهلاكها المنخفض نسبياً للذاكرة مقارنة ببدائل أخرى جرى تقييمها، وهو اعتبار حاسم في بيئة محدودة الموارد كالتي يعمل عليها هذا المشروع، دون التضحية بجودة الاستعلام والتصفية المطلوبة لعرض التنبيهات في لوحة التحكم بكفاءة.

## 7.5- تصميم الية إعادة التدريب

يوضّح الشكل (12) تصميم آلية إعادة تدريب النموذج، المصمَّمة لتمكين النظام من التكيّف تدريجياً مع أنماط حركة مرورية جديدة قد لا يكون النموذج الحالي — المدرَّب على مجموعة بيانات CSE-CIC-IDS2018 التاريخية الثابتة — قد رآها من قبل، دون الحاجة لإعادة بنائه يدوياً من الصفر في كل مرة.




الشكل 12:تصميم مخطط إعادة تدريب النماذج

تبدأ الحلقة بمجرد أن يُصدِر النموذج تنبؤه على حركة المرور الحية، حيث تُؤرشَف هذه التنبؤات بشكل دوري إلى HDFS بصيغة Parquet، لتشكّل بذلك سجلاً تراكمياً موثوقاً لكل ما "رآه" النظام فعلياً أثناء التشغيل، لا لمجرد ما وُصنِّف كتنبيه فقط. وبالتوازي، تظهر التنبيهات الناتجة عن الحركة المشبوهة في لوحة التحكم ليراجعها المحلل الأمني، الذي يُصدِر قراره النهائي بشأن كل تنبيه بوصفه إيجابياً حقيقياً أو كاذباً.

وهنا تكمن الخطوة الجوهرية في هذه الحلقة: فقرار المحلل لا يُسجَّل بمعزل عن تنبؤ النموذج، بل يُقارَن به مباشرة، إذ يُمثّل هذا القرار البشري الحكم المرجعي (Ground Truth) الذي يُقاس عليه مدى إصابة النموذج أو خطئه في كل حالة. ومع تراكم عدد كافٍ من هذه المقارنات، يتشكّل مؤشر دقة حقيقي يعكس أداء النموذج على بيانات واقعية حديثة، لا على مجموعة بيانات تدريب تاريخية ثابتة فحسب.

وحين ينخفض هذا المؤشر عن مستوى مقبول، أو يتراكم عدد كافٍ من الحالات المراجَعة،

التوصية القائمة على المحلل البشري (Human-in-the-Loop Recommendation)

لا تنطلق عملية إعادة التدريب تلقائياً بمجرد تراكم بيانات جديدة، بل تمر أولاً بمرحلة توصية يُصدِرها النظام عند تحقّق أحد شرطين: انخفاض دقة النموذج المؤكَّدة من قِبَل المحلل الأمني (Analyst-Confirmed Precision) عن عتبة 85%، أو تجاوز عدد التنبيهات المعلَّقة عتبة 200 تنبيه. ويبقى القرار النهائي بإطلاق العملية فعلياً بيد المحلل البشري لا النظام ذاته، وهو خيار تصميمي مقصود يحمي من احتمال انحراف النموذج (Model Drift) في حال كانت مراجعات المحلل نفسها غير كافية إحصائياً، إذ يشترط النظام أيضاً حداً أدنى من عدد المراجعات المكتملة قبل السماح بظهور التوصية أصلاً.

تطلَق عملية إعادة تدريب النموذج على العنقود الموزَّع، مستفيدة من أرشيف الحركة الفعلية المخزَّن في HDFS. ويخضع النموذج الناتج بعد التدريب لمقارنة مباشرة بالنموذج الحالي على معايير الأداء نفسها، لتُتَّخذ بناءً عليها إحدى نتيجتين: ترقية النموذج الجديد ليحل محل الحالي إن أثبت تفوقاً حقيقياً، أو الاحتفاظ بالنموذج القائم دون تغيير إن لم يتحقق ذلك التفوق. وبهذا، تُغلَق الحلقة لتبدأ من جديد مع كل دفعة تنبؤ لاحقة، مانحةً النظام قدرة تدريجية على التكيّف مع أنماط الهجوم المتطورة دون تدخل يدوي شامل في كل مرة.

# الفصل السادس

# مناقشة النتائج

*نستعرض في هذا الفصل التجارب التي أدت إلى بناء نموذج الذكاء الاصطناعي المعتمد، ونعرض واجهات النظام المقترح ونتائج تشغيله الفعلي.*

## 1.6- تقييم أثر توزيع البيانات على كفاءة التدريب و الإنتاجية

بهدف التحقق العلمي والدقيق من مدى جدوى وكفاءة اعتماد بيئة الحوسبة الموزعة المدارة بواسطة Apache Spark في هذا المشروع، وعزل أي عوامل برمجية أو بيئية قد تؤثر على صحة المقارنة .

تم إجراء عدة تجارب باستخدام خوارزميات مختلفة في البيئتين :

- (البيئة الموزعة): تم فيها تشغيل وتدريب النموذج الهرمي (Hierarchical Pipeline) بالاعتماد على عنقود Apache Spark موزع وممتد على ثلاث عقد فعلية (3 Nodes Distributed Cluster).

  تتشابه عقد العنقود الثلاث (k1، k2، k3) في مواصفاتها الفيزيائية، إذ تعمل كل منها كآلة افتراضية (Virtual Machine) مستضافة عبر بيئة OpenStack باستخدام تقنية الافتراضية KVM، وتمتلك معالجاً من نوع Intel Broadwell بثمانية أنوية افتراضية (8 vCPU)، وذاكرة وصول عشوائي (RAM) بسعة 16 GiB، ومساحة تخزين إجمالية قدرها 200 GB، وتعمل جميعها بنظام التشغيل Ubuntu 24.04.4 LTS بنواة Linux 6.8.0. هذا التماثل في المواصفات بين العقد الثلاث لم يكن اعتباطياً، بل قراراً مقصوداً يسهّل عملية موازنة الحمل بينها ويجعل أي عقدة قادرة على استيعاب أعباء عقدة أخرى بالكفاءة ذاتها عند الحاجة، وهو ما يخدم مباشرة متطلب قابلية التوسّع وتحمّل الأعطال الذي بُنيت عليه البنية الموزّعة للنظام.

- (البيئة المركزية المستقلة): تم فيها عزل بيئة التوزيع بالكامل، وتدريب النموذج الهرمي على عقدة مركزية واحدة (Single-Node) دون الاستعانة بـ Spark، وذلك باستخدام مكتبات المعالجة التقليدية النظيفة (Pandas و Scikit-Learn).

لضمان أعلى معايير العدالة الفنية في المقارنة المباشرة، تم تثبيت كافة المتغيرات الجوهرية عبر التجارب الثلاث، بما في ذلك حجم عينة البيانات المفرزة والمنظفة (25% من الداتا سيت الكلية لـ CIC-IDS2018 وبمجموع 4,038,551 سطر)، ونسبة فصل بيانات التدريب والاختبار (20/80)، بالإضافة إلى مطابقة المعاملات الفائقة (Hyperparameters) للنماذج .

- **التجربة الأولى — خوارزمية الغابة العشوائية (Random Forest)**

1. الإطار النظري للخوارزمية

تُعدّ خوارزمية الغابة العشوائية إحدى أبرز خوارزميات التعلّم الجماعي (Ensemble Learning)، وتقوم فكرتها الجوهرية على بناء عدد كبير من أشجار القرار (Decision Trees) المستقلة عن بعضها، ثم دمج مخرجات هذه الأشجار للوصول إلى قرار تصنيفي نهائي أكثر دقة واستقراراً من قرار أي شجرة منفردة. تعتمد الخوارزمية على تقنيتين أساسيتين لضمان تنوّع الأشجار وتقليل الارتباط بينها:

- التجميع الإحيائي (Bootstrap Aggregating – Bagging): حيث تتدرّب كل شجرة على عينة عشوائية مُعاد سحبها (Sampling with Replacement) من مجموعة البيانات الأصلية، بحيث لا ترى شجرتان نفس البيانات بالضبط.
- العشوائية في اختيار الميزات (Random Feature Selection): إذ لا تُقيَّم كل الخصائص عند كل عقدة تفرّع، بل تُختار مجموعة فرعية عشوائية منها فقط، مما يمنع هيمنة خاصية واحدة قوية على بنية جميع الأشجار ويزيد من تنوّعها.

وعند تصنيف عيّنة جديدة، تمرّ هذه العيّنة عبر جميع الأشجار في الغابة، وتُصوِّت كل شجرة بشكل مستقل على الفئة التي تراها مناسبة، ثم تُحسم النتيجة النهائية بالأغلبية (Majority Voting). وتتميز هذه الآلية بمقاومة عالية لمشكلة الإفراط في التخصيص (Overfitting) مقارنة بالشجرة المفردة، نظراً لأن الخطأ العشوائي لشجرة واحدة يُعوَّض عادة بتصويت بقية الأشجار.

2. ضبط بيئة المقارنة

لضمان أن أي فروقات تظهر في النتائج تُعزى حصراً إلى أثر البنية الحوسبية (موزعة مقابل مركزية) لا إلى اختلاف في إعدادات النموذج، تم تثبيت المعاملات الفائقة (Hyperparameters) الخاصة بالخوارزمية بشكل مطابق تماماً في كلتا التجربتين، حيث اعتُمد عدد 100 شجرة ( = n_estimators 100) وعمق أقصى يبلغ 20 (max_depth = 20)، سواء في تجربة العقدة المركزية المستقلة دون Spark، أو في تجربة العنقود الموزع على ثلاث عقد.

وللتحقق عملياً من أن التوزيع الحوسبي المعتمَد في التجربة الثانية كان فعلياً وليس نظرياً، يوثّق سجلّ تشغيل العنقود الموضّح في الشكل (X) لحظة تشكّل موارد Spark أثناء بدء المهمة:

```
Executor added: app-.../0 on worker-...-10.0.0.18-39000 (10.0.0.18:39000) with 2
core(s)
Executor added: app-.../1 on worker-...-10.0.0.124-39000 (10.0.0.124:39000) with 2
core(s)
```

الشكل 13:مخرجات سجلات التشغيل لتوزيع المنفذات (**Spark Executors**) على عقد العمل الموزعة (**k2** و **k3**).

يُظهر الشكل(13) انضمام العقدة k3 (10.0.0.18) إلى العنقود بصفتها Executor 0 بنواتين معالجة (cores 2)، وانضمام العقدة k2 (10.0.0.124) بصفتها Executor 1 بنفس السعة الحوسبية. ويُلاحَظ أن العقدة k1 (10.0.0.15) لم تُدرَج في هذا السجلّ كمُنفِّذ، لأنها تشغل دور الـ Driver المسؤول عن تنسيق المهمة (Job Scheduling) وتوزيع أشجار القرار على العقدتين المنفذتين، دون أن يُسنَد إليها عبء حسابي

مباشر. وهذا معطى منهجي مهم عند تفسير نتائج التسريع لاحقاً، إذ إن القدرة الحسابية الفعلية للعنقود تنحصر في 4 أنوية معالجة فقط (نواتان من كل عقدة منفِّذة)، وليس في عدد العقد الثلاث مجتمعة.

3. نتائج المقارنة

يلخّص الجدول (X) النتائج المقارنة بين العنقود الموزع (Spark) والعقدة المركزية المستقلة (Sklearn) عبر محاور الزمن والدقة والاستدلال:

| المقياس | 3-Node Cluster | Single-Node (sklearn | التحليل |
|---|---|---|---|
| وقت التدريب الكلي | 1882.08s | **2192.36s** (2080.78+111.58) | **Spark أسرع بـ s310 (~14.2%)** |
| وقت تدريب المرحلة الاولى | 1637.80s | 2080.78s | Spark أسرع بـ s443 (~21.3%) |
| وقت تدريب المرحلة الثانية | 244.28s | 111.58s | Single-node أسرع هون |
| الدقة | 97.75% | 97.99% | متقارب جداً |
| F1-score | 97.68% | 97.92% | متقارب |
| Precision | 97.98% | 97.91% | متقارب جداً |
| Recall | 97.75% | 97.99% | متقارب جداً |
| وقت الاستدلال الكلي | 54 s | 12 s | Spark ابطا |
| الانتاجية (Throughput) | 13,721 rows/s | 65,213 rows/s | Spark ابطأ |

4. تحليل زمن التدريب: أثر حجم البيانات على جدوى التوزيع

تكشف نتائج التدريب أن كفاءة الحوسبة الموزعة ليست مطلقة، بل مشروطة بحجم البيانات المعالَجة في كل مرحلة.

ففي المرحلة الأولى، حيث بلغ حجم بيانات التدريب أكثر من 3.2 مليون سطر، أظهر Spark تفوقاً واضحاً بفارق 443 ثانية (نسبة تحسّن 21.3%). ويُعزى هذا التفوق إلى أن العنقود الموزع استطاع تقسيم عملية بناء الأشجار المئة على العقدتين المنفذتين (k2 وk3)، بحيث نُفّذت الحسابات بالتوازي عبر الأنوية الأربع المتاحة إجمالاً، بينما اضطرت Sklearn، المقيدة بنواة واحدة (n_jobs=1) على عقدة واحدة، إلى بناء الأشجار المئة بالتتالي (Sequentially)، وهو ما انعكس مباشرة في الزمن المسجَّل (2080.78 ثانية).

```
ubuntu@k1:~/ids-training$ cat  singlenode_rf_results.json
{
  "experiment": "SingleNode_RF_sklearn_25pct",
  "mode": "single-node (scikit-learn, n_jobs=1)",
  "sample_size": 4038551,
  "train_size": 3230840,
  "test_size": 807711,
  "stage1_binary": {
    "accuracy": 0.9855418584122291,
    "f1": 0.985528214072098,
    "precision": 0.9855167782392741,
    "recall": 0.9855418584122291,
    "TP": 130854,
    "TN": 665179,
    "FP": 5515,
    "FN": 6163,
    "train_time_sec": 2080.7839081287384,
    "inference_time_sec": 10.988358974456787,
    "throughput_rows_per_sec": 73506.0623590457
  },
```

الشكل 14:نتائج تدريب النموذج التقليدي على عقدة واحدة باستخدام خوارزمية الغابات العشوائية

```
ubuntu@k1:~/ids-training$ cat distributed_rf_results
{
  "experiment": "Distributed_RF_pyspark_25pct",
  "mode": "distributed-cluster (pyspark, n_nodes=3)"
  "cluster_config": "3-Node Cluster",
  "ample_size": 4038551,
  "tain_size": 3230840,
  "test_size": 807711,
  "metrics": {
    "total_train_time_sec": 1882.08,
    "stage1_train_time_sec": 1637.80,
    "stage2_train_time_sec": 244.28,
    "accuracy": 0.9775,
    "f1": 0.9768,
    "precision": 0.9798,
    "recall": 0.9775,
    "total_inference_time_sec": 54,
    "throughput_rows_per_sec": 13721
  }
}
```

الشكل 15:نتائج تدريب النموذج على العنقود الموزع باستخدام خوارزمية الغابات العشوائية

غير أن هذا التفوق انعكس تماماً في المرحلة الثانية، حيث تقلّص حجم بيانات التدريب إلى نحو 550 ألف سطر فقط (بيانات الهجمات المفرزة دون الحركة الطبيعية). فقد استغرق Spark في هذه المرحلة 244.28 ثانية مقابل 111.58 ثانية فقط لـ Sklearn، أي أن الأخير كان أسرع بأكثر من الضعف. ويُفسَّر هذا الانعكاس بأن الفائدة الحسابية الناتجة عن التوازي أصبحت، عند هذا الحجم الأصغر، أقل من التكلفة الإضافية المرتبطة بالتوزيع ذاته (Distribution Overhead) — والمتمثلة في بث البيانات وأشجار القرار عبر الشبكة بين العقد (Network Broadcasting)، وتنسيق المهام بواسطة الـ Driver على العقدة k1، والتزامن بين الـ Executors. بالمقابل، احتفظت Sklearn بكامل البيانات في الذاكرة العشوائية (RAM) المحلية دون أي حاجة لاتصال شبكي، فكانت العملية بأكملها أسرع بطبيعتها — ولنفس هذا السبب بالضبط تفوّقت Sklearn لاحقاً في زمن مرحلة الاستدلال أيضاً، كما سيُفصَّل في المحور التالي.

```
"stage2_multiclass": {
  "accuracy": 0.9664056284986534,
  "f1": 0.9649732917819964,
  "precision": 0.9670181218946663,
  "recall": 0.9664056284986534,
  "attack_classes": [
    "Bot",
    "Brute Force -Web",
    "Brute Force -XSS",
    "DDOS attack-HOIC",
    "DDOS attack-LOIC-UDP",
    "DDoS attacks-LOIC-HTTP",
    "DoS attacks-GoldenEye",
    "DoS attacks-Hulk",
    "DoS attacks-SlowHTTPTest",
    "DoS attacks-Slowloris",
    "FTP-BruteForce",
    "Infilteration",
    "SQL Injection",
    "SSH-Bruteforce"
  ],
  "train_time_sec": 111.57810139656067,
  "inference_time_sec": 1.414862871170044,
  "throughput_rows_per_sec": 96841.18708033631
},
"total_training_time_sec": 2192.362009525299,
"unified_pipeline": {
  "routed_to_stage2": 136369,
  "stopped_at_stage1": 671342,
  "total_inference_time_sec": 12.385688066482544,
  "unified_throughput_rows_per_sec": 65213.2522363276
  "latency_ms_per_row": 0.015334306535979508,
  "accuracy": 0.9798603708504651,
  "weighted_f1": 0.9791574187885194,
  "weighted_precision": 0.9790761908957585,
  "weighted_recall": 0.9798603708504651
```


الشكل 16: نتائج تدريب المرحلة الثانية من النموذج المدرب على العنقود الوزع باستخدام خوارزمية الغابات العشوائية

تُرسّخ هذه النتيجة مبدأً هندسياً جوهرياً: التوزيع مفيد فقط حين تتجاوز تكلفة الحوسبة المتوازية تكلفة النقل والتنسيق، وهو ما يتوافق مع ما يُعرف بـ Amdahl's Law من حيث محدودية العائد من التوازي عند صغر حجم المهمة المعالَجة.

5. تحليل مقاييس الدقة: التكافؤ الخوارزمي

على الرغم من الفروقات الزمنية الملحوظة، أظهرت جميع مقاييس الأداء التصنيفي (Accuracy، F1-Score، Precision، Recall) تقارباً شديداً بين البيئتين، بفروقات لا تتجاوز 0.3% في أي مقياس. وهذا التقارب متوقَّع نظرياً ومنطقي، إذ إن كلا التطبيقين (Spark MLlib وScikit-learn) ينفذان نفس الخوارزمية الرياضية للغابة العشوائية بنفس المعاملات الفائقة على نفس عينة البيانات، والفرق الوحيد بينهما هو آلية التنفيذ الحوسبي (موزعة أم مركزية) وليس المنطق الإحصائي للنموذج ذاته.

تؤكد هذه النتيجة أن اختيار البيئة الحوسبية لا يؤثر على الجودة التصنيفية للنموذج، بل ينحصر أثره في الكفاءة الزمنية والتشغيلية فقط، وهو ما يوجّه معيار المفاضلة بين البيئتين نحو اعتبارات الأداء والموارد بدلاً من اعتبارات الدقة.

6. تحليل مرحلة الاستدلال (Inference):

بهذه التجربة يتعزز مبدأ الأنظمة الموزعة ليست أسرع دوماً في جميع مراحل التشغيل. فقد سجّل Spark زمن استدلال إجمالي بلغ 54 ثانية بإنتاجية 13,721 صف/ثانية، في حين أنجزت Sklearn نفس المهمة في 12 ثانية فقط بإنتاجية 65,213 صف/ثانية — أي أن الحل المركزي كان أسرع بما يقارب 4.75 ضعف.

ويُعزى هذا التباين إلى ذات السبب الذي فُسِّرت به نتائج المرحلة الثانية من التدريب: فعملية الاستدلال تُعالج دفعة بيانات محدودة الحجم نسبياً، وهي عملية "خفيفة حسابياً" مقارنة ببناء 100 شجرة من الصفر، ما يجعل التكلفة الثابتة لتنسيق العنقود الموزع — من بدء المهمة عبر الـ Driver، وتوزيعها على الـ Executors، وتجميع النتائج مجدداً — عبئاً نسبياً أكبر من زمن الحساب الفعلي نفسه. بينما تُنفِّذ Sklearn الاستدلال مباشرة داخل نفس عملية الذاكرة (In-Process Memory) دون أي تسلسل بيانات (Serialization) أو نقل عبر الشبكة.

7. الربط بين نتائج الأداء الزمني ونتائج مصفوفة الارتباك

من الجدير بالإشارة أن التكافؤ شبه التام في مقاييس الدقة بين البيئتين يعني أن التحديات التصنيفية المرصودة في مصفوفة الارتباك — كضعف كشف الأصناف النادرة جداً (SQL Injection، Brute Force-Web) وظاهرة الارتباك المتبادل المتناظر بين FTP-BruteForce وDoS-SlowHTTPTest — هي تحديات متعلقة ببنية البيانات ذاتها وتوزيع الفئات (Class Imbalance)، وليست نتيجة لاختيار بيئة التنفيذ الموزعة أو المركزية. وهذا يعزز التوصية بضرورة الانتقال نحو خوارزميات معالجة تسلسلية للأخطاء ،بالتوازي مع تحسين هندسة الخصائص ( Feature Engineering)، بغض النظر عن البيئة الحوسبية المعتمدة.

تخلص هذه التجربة إلى أن جدوى اعتماد Apache Spark كبيئة موزعة ليست مطلقة بل مشروطة بمرحلة المعالجة وحجم البيانات المصاحب لها: ففي مرحلة التدريب على البيانات الضخمة يُثبت Spark جدواه بتحقيقه تسريعاً يتجاوز 21%، بينما تتفوق البيئة المركزية في مرحليَّ التدريب على البيانات الأصغر حجماً والاستدلال اللحظي.

**تحليل مصفوفة الارتباك على مستوى الأصناف الفردية في التجربة الموزعة**

لاستكمال التقييم الكمي العام للنموذج (Accuracy، F1-Score) بتحليل نوعي أعمق يكشف سلوك النموذج تجاه كل نوع هجوم على حدة، جرى استخراج مصفوفة الارتباك التفصيلية على مستوى الأصناف الأربعة عشر، موضّحةً عدد الحالات الإيجابية الصحيحة (TP)، والسلبية الخاطئة (FN)، والإيجابية الخاطئة (FP) لكل صنف:

| نوع الهجوم | الإيجابي الصحيح (TP)(تم كشفه صح) | السلبي الخاطئ (FN)(هجوم لم يُكشف) | الإيجابي الخاطئ (FP)(نوع آخر شُخّص به) | الحالة والتحليل الفني |
|---|---|---|---|---|
| **DDOS attack-HOIC** | 34,437 | 1 | 3 | **ممتاز جداً** (شبه كامل) |
| **DDoS attacks-LOIC-HTTP** | 28,665 | 44 | 0 | **ممتاز جداً** |
| **DoS attacks-Hulk** | 23,468 | 0 | 1 | **مثالي** (كشف كامل بدون أخطاء) |
| **Bot** | 14,353 | 4 | 0 | **ممتاز جداً** |
| **SSH-Bruteforce** | 9,189 | 2 | 0 | **ممتاز جداً** |
| **Infiltration** | 7,869 | 76 | 7 | **جيد جداً** |
| **DoS attacks-GoldenEye** | 2,043 | 1 | 1 | **ممتاز** |
| **DoS attacks-Slowloris** | 532 | 0 | 0 | **مثالي** |
| **DDOS attack-LOIC-UDP** | 94 | 0 | 39 | **مقبول** (يوجد تداخل مع هجمات UDP أخرى) |
| **Brute Force - Web** | 26 | 3 | 77 | **ضعيف** (بسبب قلة البيانات الشديدة) |
| **Brute Force - XSS** | 11 | 2 | 4 | **مقبول نسبياً** بالنظر لحجم الكلاس |

| | | | | |
|---|---|---|---|---|
| **SQL Injection** | 4 | 3 | 2 | **غير مستقر** (العينات صغيرة جداً) |
| ● **FTP-BruteForce** | **4,024** | **5,730** | 183 | **مشكلة ارتباك حاد** (أغلب الهجمات ضاعت كـ FN) |
| ● **DoS attacks-SlowHTTPTest** | **6,799** | 181 | **5,730** | **مشكلة تداخل** (الـ FP المرتفع سببه هجمات FTP) |

أولاً — الأداء شبه المثالي في الكلاسات الغنية بالبيانات

أظهر النموذج كفاءة تصنيفية استثنائية بلغت حد الكمال في كشف هجمات حجب الخدمة والاختراق الآلي، وعلى رأسها DoS attacks-Hulk وDoS attacks-Slowloris وDDOS attack-HOIC. ويُعزى هذا الأداء إلى أن وفرة عينات التدريب المتاحة لهذه الأصناف مكّنت الغابة العشوائية من بناء حدود فصل رياضية (Decision Boundaries) متينة وواضحة، إذ استطاعت الأشجار المتعددة استخلاص الأنماط التكرارية للتدفقات الشبكية الخاصة بكل هجوم — كأزمنة الوصول ومعدلات الحزم — بدقة عالية ودون تشتت إحصائي يُذكر.

ثانياً — معضلة الأصناف النادرة (Minority Class Suppression)

في المقابل، سجّل النموذج تراجعاً واضحاً في استقراره التصنيفي عند التعامل مع الهجمات الحرجة الأقل تمثيلاً في عينة التدريب، وعلى رأسها SQL Injection وBrute Force - Web. يُفسَّر هذا التراجع علمياً بالطبيعة الإحصائية لخوارزميات الغابة العشوائية التقليدية، التي تُحسَّن أساساً لتقليل معدل الخطأ الإجمالي للنموذج (Global Error Minimization) بدلاً من ضمان عدالة الأداء عبر الأصناف (Per-Class Fairness). ونتيجة لذلك، تميل الخوارزمية إحصائياً إلى تهميش الأصناف التي لا تمثل سوى أجزاء من الألف من إجمالي العينات لصالح الأصناف المهيمنة عددياً، مما يحول دون تكوّن فروع تقسيمية كافية داخل الأشجار لتمييز هذه الهجمات بثقة كافية عند التشغيل الفعلي — وهي نتيجة متوقعة نظرياً وتُعرف في أدبيات التعلم الآلي بمشكلة Class Imbalance Bias.

ثالثاً — ظاهرة الارتباك المتبادل المتناظر (Symmetric Cross-Confusion)

يُمثّل الارتباك الحاد المرصود بين هجومي FTP-BruteForce وDoS attacks-SlowHTTPTest ، إذ تكشف مصفوفة الارتباك عن تطابق رقمي دقيق ولافت: فعدد العينات المفقودة من هجوم FTP-BruteForce (FN = 5,730) يُطابق تماماً عدد العينات المصنَّفة زيفاً ضمن هجوم SlowHTTPTest (FP = 5,730).

هذا التطابق الرقمي التام ليس صدفة إحصائية، بل دليل هندسي قاطع على وجود تداخل حقيقي في فضاء الخصائص (Feature Space Overlap) بين الهجومين عند مستوى الميزات المستخرجة بواسطة CICFlowMeter — وتحديداً في الخصائص المرتبطة بأزمنة وصول الحزم (Inter-Arrival Time) وأطوال التدفقات (Flow Duration)، والتي تتشابه سلوكياً بين النمطين نظراً لكون كليهما هجوماً بطيء المعدل (Low-and-Slow Attack). ولما كان حجم كلاس SlowHTTPTest أكبر عددياً في بيانات التدريب مقارنة بـ FTP-BruteForce، مالت الخوارزمية "رياضياً" إلى توجيه التدفقات المتشابهة نحو الصنف الأكبر حجماً، بوصفه الخيار الأقل كلفة من حيث دالة الخطأ الإجمالية.

- **التجربة الثانية — خوارزمية التدرج المعزز (Gradient Boosting Trees)**

1. الإطار النظري للخوارزمية

تُعدّ خوارزمية أشجار التعزيز التدرجي (Gradient Boosted Trees - GBT) واحدة من أكثر خوارزميات التعلم الآلي كفاءةً ودقة في التعامل مع البيانات الهيكلية، وهي تنتمي إلى فئة التعلم التجميعي (Ensemble Learning) القائم على أسلوب التعزيز (Boosting).

على عكس خوارزمية الغابات العشوائية (Random Forests) التي تبني أشجار القرار بشكل مستقل ومتوازٍ، تعمل GBT وفق منهجية تتابع خطية حيث يتم بناء سلسلة من أشجار القرار الضعيفة. تبدأ الخوارزمية بنموذج أولي بسيط، ثم تُبنى كل شجرة جديدة خصيصاً للتنبؤ بـ البواقي (Residuals) — وهي الفروقات والأخطاء الناتجة عن التنبؤات التجمعيّة للأشجار السابقة.

تعتمد الخوارزمية في تحسين أدائها على التقليل المستمر لـ دالة الخسارة (Loss Function) باستخدام طريقة الهبوط التدريجي (Gradient Descent)، حيث يمثل تدرج دالة الخسارة الاتجاه الذي يجب أن تتخذه الشجرة التالية لتصحيح التنبؤ. ويتم التحكم بمدى مساهمة كل شجرة جديدة من خلال معامل معدل التعلم (Learning Rate / Step Size) لضمان استقرار النموذج ومنع الإفراط في التكيف (Overfitting).

- **استغلال التوازي والبيانات الضخمة في Apache Spark (Distributed GBT Optimization)**

على الرغم من الطبيعة المتسلسلة المضمنة في خوارزمية GBT — والتي تشكل عائقاً تقليدياً أمام التوازي الكامل — إلا أن مكتبة التعلم الآلي الموزع Spark MLlib (عبر الصنف GBTClassifier) قدمت معالجة مبتكرة تمكنها من الاستفادة القصوى من البنية الموزعة لبيئات البيانات الضخمة (Clusters).

تعتمد Spark MLlib في تحقيق التوازي وأعلى مستويات الأداء على المحاور التالية:

- التوزيع الأفقي للبيانات (Distributed Data Processing):

تتعامل المكتبة مع مصفوفات البيانات الضخمة المقسمة عبر الـ Executors ضمن الـ Cluster باستخدام البنى الموزعة (DataFrames / RDDs). تظل البيانات محتفظاً بها في الذاكرة المؤقتة (In-Memory Pipeline)، مما يلغي بطء عمليات القراءة والكتابة من القرص الصلب أثناء تكرارات بناء الأشجار.

- التوازي في مستوى الشجرة الواحدة (Parallel Tree Splitting):

عند بناء كل شجرة جديدة في السلسلة، لا ينتظر Spark الخادم الرئيسي لحساب أفضل نقاط التقسيم (Splitting Points) للميزات (Features). بل يوزع عمليات حساب المدرجات التكرارية (Histograms) وتجميع الإحصاءات للميزات بالتوازي بين جميع العُقد (Worker Nodes)، ومن ثم تُجمع النتائج مركزياً لاختيار التقسيم الأمثل لكل نقطة تفريع (Node Split).

- التكامل مع خطوط المعالجة التدفقية (Streaming Pipeline Integration):

تتميز مكتبة GBT في Spark بتكاملها المباشر مع خطوط معالجة الميزات (Pipeline APIs مثل VectorAssembler و StringIndexer). هذا يتيح تطبيق النموذج المدرّب بشكل لحظي ومباشر على الدفعات الدقيقة (Micro-batches) القادمة من محرك Spark Streaming، مما يمنح النظام القدرة على تصنيف ترافيك الشبكة واستخراج التنبيهات بزمن تأخير متدنٍ جداً (Sub-second Latency) وتحت أحمال تدفق مرتفعة.

2. الجدول الأول — المقاييس الزمنية والتصنيفية

يلخّص الجدول التالي النتائج المقارنة بين بيئة Spark الموزعة على ثلاث عقد والبيئة المركزية المستقلة (Scikit-learn)، باستخدام نفس حجم العينة الموحّد (25% من CIC-IDS2018) ونفس منهجية الفصل الهرمي (Binary ثم Multiclass) المعتمدة في التجربة الأولى:

| المقياس / مرحلة الاختبار | بيئة Spark الموزعة (3 عُقد) | البيئة المركزية (Scikit-Learn) | نسبة التفوق / الفارق الحسابي |
|---|---|---|---|
| **زمن تدريب المرحلة الأولى (Binary)** | **791.56. ثانية (~13.5 دقيقة)** | 8,127.70 ثانية (~135.5 دقيقة) | **Spark أسرع بـ 10.07 مرات** |
| **زمن تدريب المرحلة الثانية (Multiclass)** | **938.54 ثانية (~15.6 دقيقة)** | 4,252.48 ثانية | **Spark أسرع بـ 4.53 مرات** |
| **إجمالي زمن تدريب النظام (Total Train)** | **1,746.11 ثانية (~29.1 دقيقة)** | 12,380.18 ثانية (~3.4 ساعات) | **اختصار زمن التدريب بـ 7 مرات** |
| **زمن استدلال المرحلة الأولى (Binary)** | 7.83ثانية | 2.35ثانية | العقدة المفردة أسرع بـ 3.62 مرات |
| **زمن استدلال المرحلة الثانية (Multiclass)** | 15.77ثانية | **1.74ثانية** | **العقدة المفردة أسرع بـ 6.13 مرات** |
| **إجمالي زمن الاستدلال الموحد (Unified)** | 23.60ثانية | **3.96ثانية** | العقدة المفردة أسرع بـ 7.94 مرات |
| **الدقة الكلية الموحدة ( Unified Acc)** | **%98.28** | %98.22 | متقارب إحصائياً |
| **F1-Score الموحد للنظام** | **%98.13** | %98.10 | متقارب إحصائياً |
| **إنتاجية الاستدلال الموحدة (Throughput)** | 25,712.5 سجل/ثانية | **204,022.8 سجل/ثانية** | **العقدة المفردة أسرع بـ 7.94 مرات** |

3. تحليل زمن التدريب:

يكشف الجدول عن قفزة نوعية في أثر التوزيع الحوسبي مقارنة بما شُوهد سابقاً في تجربة الغابة العشوائية، إذ لم يقتصر تفوق Spark هذه المرة على نسبة محدودة (~21% في RF)، بل بلغ 10.07 مرة في المرحلة الأولى وحدها. ولفهم هذا التباين الحاد بين التجربتين، لا بد من العودة إلى الفرق البنيوي الجوهري بين الخوارزميتين الموضَّح في الإطار النظري أعلاه:

ففي حالة الغابة العشوائية، تُعالج بيئة Sklearn المركزية القيد بأسلوب واحد فقط — تسلسل بناء الأشجار المستقلة (n_jobs=1) — وهو قيد يمكن تعويضه جزئياً لأن كل شجرة تُبنى بسرعة نسبية معتمدة فقط على حجم عينتها الفرعية. أما في حالة GBT، فإن القيد التسلسلي مضاعف: فبالإضافة إلى استحالة بناء الأشجار بالتوازي (نظراً لاعتماد كل شجرة على مخرجات سابقتها)، يقع على عاتق المعالج الواحد في البيئة المركزية عبء إضافي هائل يتمثل في فحص نقاط التقسيم المثلى (Exact Splitting Candidates) عبر الميزات الشبكية لكل شجرة، وتكرار هذا الفحص عبر ملايين السجلات خطوة بخطوة دون أي إمكانية لتوزيع هذا العبء الحسابي داخلياً.

في المقابل، وعلى الرغم من أن Spark لا يستطيع كسر التسلسل بين الأشجار (فهذا قيد رياضي في الخوارزمية ذاتها لا يمكن لأي بنية حوسبية تجاوزه)، فإنه ينجح في توزيع العبء الحسابي داخل كل شجرة على حدة: إذ تُوزَّع السجلات الشبكية على العقد العاملة (k2 وk3) لتتحمل كل عقدة جزءاً من عملية حساب التدرجات (Gradient Computation) والاشتقاقات الإحصائية محلياً، وتُرسل النتائج الجزئية فقط إلى العقدة الرئيسية (Driver على k1) لتجميعها وبناء الشجرة التالية. هذه الآلية تكسر عنق الزجاجة الحسابي (Computational Bottleneck) الذي كان سيُصيب Spark أيضاً لو اعتمد فقط على توازي الأشجار كما في RF.

نلاحظ أن فائدة التوزيع الحوسبي ليست ثابتة عبر الخوارزميات، بل تتناسب طردياً مع درجة تعقيد وحجم العبء الحسابي داخل الوحدة الأساسية للتدريب (الشجرة الواحدة في هذه الحالة). فكلما كانت الخوارزمية أكثر اعتماداً على عمليات مكثفة حسابياً يصعب اختصارها، كلما تضاعفت الفائدة العائدة من توزيعها على عقد متعددة.

4. معضلة الاستقرار الرقمي

تكشف النتائج التجريبية عن ظاهرة حرجة تتجاوز في أهميتها مجرد الفارق الزمني، وتُمثّل أهم نتيجة نوعية في هذه التجربة بأكملها: فأثناء تدريب المرحلة الثانية (Multiclass) على العقدة المفردة، تعرّضت دالة الخسارة لانفجار رقمي متباعد (Numerical Overflow/Divergence) بدءاً من التكرار التاسع، حيث قفزت قيمة الخسارة إلى مستويات أسّية غير منطقية ($\sim 6.8 \times 10^{46}$). ويُعزى هذا السلوك إلى شدة عدم توازن البيانات (Severe Class Imbalance) في بعض الفئات النادرة جداً كـ SQL Injection، حيث أن آلية التحسين التسلسلي في GBT — التي تُعيد وزن الأخطاء المتراكمة عند كل تكرار — تصبح شديدة الحساسية لهذا النوع من العينات النادرة عندما تُعالَج دفعة واحدة على معالج مركزي دون أي آلية لتوزيع أو استقرار العينات إحصائياً.

في المقابل، حافظت خوارزمية GBT ضمن بيئة Spark على استقرارها الرقمي الكامل وتدرّبت بسلاسة إحصائية متناهية عبر جميع التكرارات دون أي مؤشر تباعد.

هذه النتيجة تحمل دلالة تتجاوز حدود المقارنة الزمنية البحتة التي سادت التجربة الأولى: فبينما كانت النتيجة الأبرز في تجربة RF أن البيئتين متكافئتان من حيث الدقة ومختلفتان فقط من حيث الزمن، تُثبت هذه النتيجة أن البيئة المركزية في حالة GBT لم تفشل زمنياً فحسب، بل أنتجت في إحدى المحاولات نموذجاً غير قابل للاستخدام إحصائياً لولا ضبط معاملات إضافية للتعامل مع الانفجار الرقمي. وبذلك، تنتقل فائدة الحوسبة الموزعة هنا من كونها "تحسيناً في الأداء" إلى كونها ضامناً أساسياً لسلامة عملية التدريب ذاتها عند التعامل مع خوارزميات تسلسلية حساسة رقمياً وبيانات شديدة التفاوت في توزيع الفئات — وهو استنتاج أعمق بكثير من مجرد مقارنة الأزمنة.

5. مفارقة كفاءة الاستدلال: تكرار نمط ثابت عبر الخوارزميتين

على النقيض التام من مرحلة التدريب، تفوّقت العقدة المفردة بشكل حاد في مرحلة الاستدلال، محقّقةً إنتاجية بلغت 204,022.8 سجل/ثانية مقابل 25,712.5 سجل/ثانية لـ Spark، أي بفارق 7.94 مرة. ويُفسَّر هذا السلوك بنفس المنطق الهندسي الذي فُسِّرت به نتيجة مماثلة في تجربة الغابة العشوائية: فعملية التنبؤ (Prediction/Transform) عملية خفيفة حسابياً مقارنة ببناء النموذج، إذ تقتصر على تمرير كل سجل عبر أشجار جاهزة مسبقاً دون أي حاجة لحساب تدرجات أو تحديث أوزان. وفي هذه الحالة، يتحوّل الحمل الإضافي الناتج عن تنسيق الشبكة بين العقد العاملة والـ Driver إلى عبء صافٍ يعيق السرعة، في مقابل الوصول المباشر للذاكرة المحلية في البيئة المركزية.

وتكرار هذا النمط بدقة في كلتا التجربتين (RF وGBT) يرفعه من مجرد ملاحظة عرضية إلى قاعدة تصميمية ثابتة يمكن البناء عليها بثقة في الفصل الخاص بالتصميم النهائي للنظام: أنظمة كشف التسلل الموزعة تُصمَّم أساساً لمواجهة معضلات التدريب الضخم وإعادة التدريب الدوري، بينما تحتاج طبقة الكشف اللحظي (Real-Time Detection Layer) إلى نشر النماذج المستخلصة (Exported Models) محلياً في نقاط الفحص الشبكي لتقليل زمن الاستجابة (Latency)، بدلاً من استدعاء عنقود Spark بأكمله لكل عملية تنبؤ فردية.

6. الجدول الثاني — التحليل الأمني: الإنذارات الكاذبة ومعدلات الإفلات

يتجاوز تقييم أنظمة كشف التسلل مقياس الدقة الإجمالية إلى تحليل أعمق لمصفوفة الارتباك من منظور أمني تشغيلي، إذ لكل نوع خطأ تصنيفي كلفة أمنية مختلفة جوهرياً. يلخّص الجدول التالي هذا التحليل لمرحلة الكشف الثنائي (Binary):

| المقياس الحسابي / الأمني | بيئة Spark الموزعة (3 عُقد) | البيئة المركزية (Scikit-Learn) | التفسير الأمني الحرج |
|---|---|---|---|
| **الحالات الإيجابية الصحيحة (TP)** | 131,042 | 131,008 | هجمات تم كشفها بنجاح (متقارب) |
| **الحالات السلبية الصحيحة (TN)** | 666,881 | 666,937 | حركة طبيعية سمح بها النظام (متقارب) |
| **الإنذارات الإيجابية الكاذبة (FP)** ***(False Positive Alarm)*** | **3,813 سجل** | **3,757 سجل** | حركة طبيعية أطلق النظام ضدها إنذاراً |
| **معدل الإنذارات الكاذبة (FPR)** | **%0.568** | **%0.560** | نسبة الإنذارات الخاطئة (أقل من 1% للطرفين ) |
| **الإنذارات السلبية المفلتة (FN)** ***(False Negative Alarm)*** | **5,975 سجل** | **6,009 سجل** | **هجمات حقيقية نجحت في اختراق النظام دون إنذار!** |
| **معدل الإفلات الأمني (FNR)** | **%4.36** | **%4.38** | نسبة الهجمات التي عبرت دون أن تُكتشف |

7. تحليل الإنذارات الكاذبة ومعدلات الإفلات:

يُظهر التحليل المقارن لمصفوفة الارتباك تطابقاً كبيراً في الكفاءة الإحصائية بين البيئتين، مع تفوق طفيف جداً للبيئة المركزية في تقليص الإنذارات الكاذبة (FP) بمقدار 56 سجلاً فقط (FPR: 0.560% مقابل 0.568%). ويُعزى هذا التقارق شبه التام إلى تطابق البنية الرياضية للخوارزمية ذاتها في كلا الجانبين، ما يجعل سلوك تحديد عتبات الفصل (Decision Thresholds) للحركة الطبيعية متماثلاً تقريباً بغض النظر عن بيئة التنفيذ — وهي نتيجة متسقة مع ما ثبت سابقاً في تجربة RF من أن اختيار البيئة الحوسبية لا يمس جوهر القرار التصنيفي للنموذج.

غير أن المحك الحقيقي لأي نظام كشف تسلل لا يكمن في مقياس FP بل في مقياس FN، إذ يمثّل كل سجل مفلت هجوماً حقيقياً نجح في تجاوز خطوط الدفاع دون أن يُثار أي إنذار بشأنه. وهنا بالتحديد يظهر التفوق الأمني الحقيقي لعنقود Spark: فقد نجح النموذج الموزع في رصد 34 هجوماً إضافياً كانت لتُفلت لو اعتمد النظام على المعالجة المركزية وحدها، مسجّلاً معدل إفلات أمني (FNR) بلغ 4.36% مقابل 4.38% للعقدة المفردة.

وعلى الرغم من أن هذا الفارق يبدو ضئيلاً من الناحية الرقمية البحتة (0.02 نقطة مئوية فقط)، فإن دلالته الأمنية أكبر بكثير من حجمه العددي في سياق بيئات الحوسبة السحابية واسعة النطاق: فكل سجل من هذه الـ34 سجلاً المفلتة قد يمثّل محاولة اختراق متقدمة — كهجوم SQL Injection ناجح أو محاولة تهديد مستمر متقدم (Advanced Persistent Threat) — تصل فعلياً إلى قواعد البيانات الحساسة دون أن يُطلق النظام أي تنبيه، وهو سيناريو الفشل الأخطر لأي نظام IDS بصرف النظر عن نسبته المئوية الظاهرية.

ويُعزى هذا التميز الطفيف لصالح Spark إلى نفس الآلية التي فسّرت تفوقه الكاسح في زمن التدريب: فتوزيع البيانات عبر تقنية الـ Partitioning يتيح حساب التدرجات وتحديث الأوزان اعتماداً على كتل بيانات موزعة تُمثِّل إحصائياً تنوّعاً أوسع من العينات في كل خطوة تحسين، بخلاف المعالجة التسلسلية الصرفة على عقدة واحدة التي قد تنحاز تدريجياً نحو الأنماط الأكثر تكراراً في تسلسل معالجتها الخطي — وهو ما يقلل من انحياز النموذج ( Model Bias) تجاه الفئات المهيمنة ويحسّن قدرته على التقاط الهجمات النادرة والمعقدة.

8. الاستنتاج المقارن بين التجربتين

عند وضع نتائج هذه التجربة إلى جانب تجربة الغابة العشوائية، تتضح صورة منهجية متكاملة تُثري نقاش المشروع بشكل جوهري: فبينما أظهرت خوارزمية RF أن فائدة التوزيع محدودة ومشروطة بحجم البيانات فقط (تفوق طفيف لا يتجاوز 21% عند البيانات الضخمة، وانعكاس كامل للصالح البيئة المركزية عند تقلّص الحجم)، تُثبت خوارزمية GBT أن هذه الفائدة تتضاعف بشكل حاد (تصل إلى 10 أضعاف) حين تكون الخوارزمية ذات طبيعة تسلسلية معقدة حسابياً، بل وتتجاوز حدود "تسريع الأداء" إلى كونها شرطاً لضمان الاستقرار الرقمي للتدريب ذاته.

كما تتفق التجربتان معاً على نمط ثابت واحد لا يتغير: تفوق البيئة المركزية الحاسم في مرحلة الاستدلال اللحظي في كلتا الخوارزميتين، وهو ما يرسّخ التوصية التصميمية النهائية للمشروع بأن تُعتمد بنية Spark الموزعة حصراً في طبقة التدريب وإعادة التدريب الدوري على البيانات التاريخية الضخمة، بينما يُنشَر النموذج النهائي المصدَّر في طبقة الكشف اللحظي عن الهجمات عبر استدلال مركزي خفيف الوزن، بما يحقق التوازن الأمثل بين قابلية التوسع وسرعة الاستجابة الأمنية الحرجة.

```
STAGE 1 — Binary (single-node, scikit-learn GradientBoosting - CLASSIC)
==================================================================
Using classic (exact-splitting) GradientBoostingClassifier to match
Spark MLlib's GBTClassifier exactly in algorithmic behavior and
hyperparameters (n_estimators=50, max_depth=10), for a fully fair
single-node vs distributed-cluster comparison.
      Iter       Train Loss   Remaining Time
         1           1.2148          122.73m
         2           1.0744          123.00m
         3           0.9576          120.96m
         4           0.8591          125.19m
         5           0.7760          122.17m
         6           0.7041          119.60m
         7           0.6419          116.91m
         8           0.5881          118.98m
         9           0.5413          117.69m
        10           0.5001          114.32m
        20           0.2752           85.13m
        30           0.2015           55.52m
        40           0.1717           27.41m
        50           0.1580            0.00s
Stage1 -> Acc=0.9879 F1=0.9879 Prec=0.9879 Rec=0.9879
Stage1 -> TP=131008 TN=666937 FP=3757 FN=6009
Stage1 -> Train time: 8127.70s | Inference time: 2.36s | Throughput: 342875.3 rows/s
Stage 1 checkpoint saved.
==================================================================
STAGE 2 — Multiclass (single-node, scikit-learn GradientBoosting - CLASSIC)
==================================================================
Attack classes: ['Bot', 'Brute Force -Web', 'Brute Force -XSS', 'DDOS attack-HOIC', 'DDOS a
S attacks-GoldenEye', 'DoS attacks-Hulk', 'DoS attacks-SlowHTTPTest', 'DoS attacks-Slowlori
jection', 'SSH-Bruteforce']
Stage2 Train: 550460 | Test: 137017
      Iter       Train Loss   Remaining Time
         1           1.2821           52.25m
         2           1.0044           66.65m
         3           0.8127           67.73m
```

الشكل 17:نتائج تديب نموذج الذكاء الصنعي على عقدة واحدة باستخدام **gpt**

```
Master: spark://10.0.0.15:7077
Default parallelism: 2
Full clean rows: 16137183
26/07/09 20:55:40 WARN SparkStringUtils: Trunc
ted by setting 'spark.sql.debug.maxToStringFie
Sample (25%) rows: 4038551
+-----------------------+-------+
|Label                  |count  |
+-----------------------+-------+
|Benign                 |3351074|
|DDOS attack-HOIC       |171425 |
|DDoS attacks-LOIC-HTTP |144106 |
|DoS attacks-Hulk       |116138 |
|Bot                    |71634  |
|FTP-BruteForce         |48422  |
|SSH-Bruteforce         |46612  |
|Infilteration          |40442  |
|DoS attacks-SlowHTTPTest|34941 |
|DoS attacks-GoldenEye  |10405  |
|DoS attacks-Slowloris  |2670   |
|DDOS attack-LOIC-UDP   |461    |
|Brute Force -Web       |146    |
|Brute Force -XSS       |53     |
|SQL Injection          |22     |
+-----------------------+-------+
```

الشكل 18 **:** بدء تدريب نموذج على العنقود الموزع باستخدام خوارزمية **gpt**

```
26/07/09 21:11:08 WARN DAGScheduler: Broadcasting large task binary with size 5.8 MiB
Stage1 -> Acc=0.9878 F1=0.9878 Prec=0.9878 Rec=0.9878
Stage1 -> TP=131276 TN=666800 FP=3700 FN=6140
Stage1 -> Train time: 791.56s | Inference time: 7.83s | Throughput: 103118.0 rows/s
```

الشكل 19: نتائج تدريب المرحلة الاولى من نموذج المدرب على العنقود الموزع باستخدام **gpt**

**التجربة الثالثة — الشبكات العصبية متعددة الطبقات (Multi-Layer Perceptron – MLP)**

1. الإطار النظري للخوارزمية

تختلف الشبكات العصبية متعددة الطبقات جذرياً عن الخوارزميتين السابقتين (RF وGBT) من حيث الأساس الرياضي للتعلّم، إذ لا تعتمد على مجموعة من أشجار القرار، بل على بنية طبقية من الوحدات العصبونية (Neurons) المترابطة، تُعالج المدخلات عبر سلسلة من التحويلات الخطية وغير الخطية المتتالية. تتكوّن الشبكة من طبقة إدخال تستقبل الخصائص الشبكية، وطبقة أو أكثر من الطبقات المخفية (Hidden Layers) التي تستخلص تمثيلات متدرجة التجريد للبيانات عبر تابع التفعيل الشرطي (ReLU)، وطبقة إخراج تستخدم تابع Softmax لتحويل المخرجات إلى توزيع احتمالي على الأصناف المحتملة.

يتم ضبط أوزان الشبكة عبر خوارزمية تحسين تكرارية تسعى لتقليل دالة الخسارة على كامل مجموعة التدريب، وفي هذه التجربة اعتُمد المحسِّن شبه النيوتني L-BFGS، وهو خوارزمية تحسين من الرتبة الثانية تستخدم تقريباً لمصفوفة الهيسيان (Hessian Approximation) لتسريع التقارب مقارنة بخوارزميات الانحدار التدرجي البسيطة، وهي خاصية مهمة عند العمل بعدد محدود من التكرارات كما هو الحال هنا (maxIter = 60).

والفرق الجوهري بين MLP والخوارزميتين السابقتين من منظور الحوسبة الموزعة يكمن في طبيعة العملية الحسابية الأساسية: فبينما تُبنى الأشجار في RF وGBT عبر عمليات منطقية متكررة للبحث عن نقاط تقسيم مثلى، تعتمد الشبكة العصبية على عمليات ضرب مصفوفي كثيف ( Dense Matrix Multiplication) بين أوزان الطبقات ومدخلات البيانات — وهي عملية ذات طبيعة حسابية مختلفة تماماً من حيث قابليتها للتوزيع وحساسيتها لزمن نقل البيانات عبر الشبكة، كما سيتضح في تحليل النتائج.

2. ضبط بيئة المقارنة والتكافؤ الإحصائي الصارم

للوصول إلى أعلى درجة ممكنة من العدالة المنهجية بين البيئتين، جرى في هذه التجربة تثبيت عدد أكبر من المتغيرات مقارنة بالتجربتين السابقتين، نظراً لحساسية الشبكات العصبية الشديدة للتهيئة الأولية للأوزان (Weight Initialization):

تطابق العينة والتقسيم: استُخدمت نفس عينة الـ25% من CIC-IDS2018 في كلتا البيئتين، مع تثبيت بذرة العشوائية (Seed = 42) واعتماد نفس نسبة التقسيم (80% تدريب / 20% اختبار).

تطابق البنية الهيكلية: بُنيت طبقات الشبكة بأحجام متطابقة رقمياً في البيئتين: طبقتان مخفيتان بحجم [40, 20] عصبون في المرحلة الأولى (Binary)، وطبقتان مخفيتان بحجم [50, 30] عصبون في المرحلة الثانية (Multiclass).

تطابق خوارزمية التحسين: اعتُمد تابع التفعيل ReLU للطبقات المخفية، وSoftmax لطبقة الإخراج، والمحسِّن L-BFGS بحد أقصى 60 تكراراً (maxIter = 60)، مع توحيد بذرة تهيئة الأوزان (Weight Initialization Seed) في الطرفين لضمان انطلاق كلا النموذجين من نفس نقطة البداية الرياضية.

التحقق من تكافؤ الموازنة الفعلي: جرى التحقق البرمجي المباشر من نتائج معالجة عدم التوازن (Class Imbalance Handling)، حيث بلغ حجم بيانات التدريب في المرحلة الأولى 5,359,649 سجلاً في بيئة Spark مقابل 5,360,760 سجلاً في البيئة المركزية — بفارق نسبي لا يتجاوز 0.02%، وهو فارق مُهمل إحصائياً يعود إلى طبيعة السحب الاحتمالي الموزع (Distributed Probabilistic Sampling) في Spark مقابل الحساب المحلي الدقيق (Deterministic Local Computation) في Sklearn، وليس إلى أي تباين منهجي حقيقي بين التجربتين.

هذا المستوى من الدقة في ضبط المتغيرات يمنح هذه التجربة، بخلاف التجربتين السابقتين، ثقة إضافية بأن أي فارق مُلاحَظ في النتائج لاحقاً يعكس أثراً حقيقياً للبنية الحوسبية لا فروقاً خفية في إعداد التجربة.

2. جدول النتائج المقارن

| المقياس الحسابي والأمني | بيئة Spark الموزعة (3 عُقد) | البيئة المركزية (Single-Node) | جهة التفوق / الفارق الأدائي |
|---|---|---|---|
| **زمن تدريب المرحلة الأولى (Binary)** | 633.24 ثانية (~10.5 دقيقة) | **286.42 ثانية (~4.7 دقيقة)** | العقدة المفردة أسرع بـ ×2.21 |
| **زمن تدريب المرحلة الثانية (Multiclass)** | 243.33 ثانية (~4.0 دقيقة) | **121.51 ثانية (~2.0 دقيقة)** | العقدة المفردة أسرع بـ ×2.00 |
| **إجمالي زمن تدريب الشبكة (Total Train)** | 876.57 ثانية (~14.6 دقيقة) | **407.93 ثانية (~6.8 دقيقة)** | **العقدة المفردة أسرع بـ ×2.15** |
| **زمن استدلال المرحلة الأولى (Binary)** | 11.24 ثانية | **0.68 ثانية** | ⚡ العقدة المفردة أسرع بـ ×16.5 |
| **زمن استدلال المرحلة الثانية (Multiclass)** | 7.35 ثانية | **0.12 ثانية** | ⚡ العقدة المفردة أسرع بـ ×61.2 |
| **إجمالي زمن الاستدلال الموحد (Unified)** | 18.59 ثانية | **0.80 ثانية** | **⚡ العقدة المفردة أسرع بـ ×23.2** |
| **دقة المرحلة الأولى ( Stage 1 Acc)** | %98.50 | %98.50 | تطابق إحصائي مطلق 🎯 |
| **دقة المرحلة الثانية ( Stage 2 Acc)** | **%95.96** | %94.93 | التفوق لصالح Spark بـ %1.03 |
| **الدقة الكلية الموحدة ( Unified Acc)** | **%97.98** | %97.77 | متقارب جداً لصالح Spark |
| **F1-Score الموحد للنظام** | **%97.89** | %97.64 | متقارب جداً لصالح Spark |
| **إنتاجية الاستدلال الموحدة (Throughput)** | 41,397.6 سجل/ثانية | **1,189,106.0 سجل/ثانية** | **العقدة المفردة أسرع بـ ×28.7** |

حليل زمن التدريب: انعكاس تام لنمط التجربة الثانية

تُظهر هذه التجربة نتيجة لافتة عمّا شُوهد في تجربتَي RF وGBT: فللمرة الأولى عبر التجارب الثلاث، تتفوق البيئة المركزية على العنقود الموزع في زمن التدريب ذاته، وليس فقط في مرحلة الاستدلال، بفارق يتراوح بين 2.00× و2.21× عبر المرحلتين.

ويُفسَّر هذا الانعكاس بالعودة إلى الفرق البنيوي المذكور في الإطار النظري: فعملية تدريب الشبكة العصبية عبر L-BFGS تعتمد على حساب متكرر لتدرجات الخسارة (Gradients) وتحديث شامل لمصفوفات الأوزان في كل تكرار، وهي عملية تتطلب تزامناً دقيقاً ومتكرراً (Frequent Synchronization) بين جميع العقد المشاركة بعد كل خطوة تحسين — بخلاف بناء شجرة القرار الواحدة في RF أو GBT، والتي وإن تطلبت تنسيقاً بين العقد، فإنها لا تحتاج لنفس الكثافة من التزامن اللحظي. وحين يكون حجم الشبكة نفسها صغيراً نسبياً (طبقات من 20 إلى 50 عصبوناً فقط) مقارنة بحجم البيانات المدارة، تصبح تكلفة التزامن المتكرر بين العقد الثلاث عبر الشبكة الداخلية أكبر من الفائدة الحسابية المكتسبة من توزيع عمليات الضرب المصفوفي، التي هي أصلاً عمليات سريعة جداً على معالج واحد حديث.

هذه النتيجة تُكمِّل الصورة المنهجية التي بدأت تتشكل عبر التجارب الثلاث: فإذا كانت التجربة الأولى (RF) قد أثبتت أن فائدة التوزيع مشروطة بحجم البيانات، وأثبتت التجربة الثانية (GBT) أنها تتضاعف مع تعقيد العملية الحسابية داخل الوحدة الأساسية للتدريب، فإن هذه التجربة (MLP) تضيف بُعداً ثالثاً حاسماً: فائدة التوزيع مشروطة أيضاً بمدى حاجة الخوارزمية للتزامن المتكرر بين العقد أثناء التدريب. فكلما زاد عدد نقاط التزامن الإلزامية (كما في التحديث التكراري لأوزان شبكة عصبية بأكملها)، زادت حساسية الخوارزمية لزمن نقل البيانات، حتى لو كان حجم البيانات الإجمالي ضخماً كما هو الحال هنا (فوق 5.3 مليون سجل).

5. تحليل مرحلة الاستدلال: أشد فجوات الأداء عبر التجارب الثلاث

يُمثّل الفارق المسجَّل في هذه التجربة بين البيئتين في مرحلة الاستدلال — 28.7 مرة لصالح العقدة المفردة، بإنتاجية تجاوزت 1.18 مليون سجل في الثانية — أضخم فجوة أداء رُصدت عبر التجارب الثلاث مجتمعة (مقارنة بـ4.75× في RF و7.94× في GBT). ويُفسَّر هذا التفاوت الحاد هندسياً بأن عملية الاستدلال في الشبكات العصبية تنحصر في تمرير أمامي واحد (Single Forward Propagation) عبر طبقات قليلة العدد وصغيرة الحجم نسبياً — وهي أخف عملية حسابية عبر جميع الخوارزميات الثلاث المختبَرة في هذا المشروع. وعند هذا المستوى من الخفة الحسابية، يُصبح أي حمل شبكي إضافي (Network Overhead) ناتج عن تسلسل البيانات (Serialization) وخلطها (Shuffling) بين الـ Driver والـ Workers مهيمناً بالكامل على الزمن الإجمالي، لدرجة أن الفائدة النظرية للتوازي تتلاشى تماماً وتتحوّل إلى عبء صافٍ يفوق زمن الحساب الفعلي بعشرات المرات.

وبذلك، فإن تكرار نمط "تفوق العقدة المفردة في الاستدلال" عبر الخوارزميات الثلاث كافة (RF، GBT، MLP) — مع تصاعد حدة هذا التفوق كلما خفّت الكثافة الحسابية للنموذج (من 4.75× في RF الأثقل نسبياً، إلى 28.7× في MLP الأخف) — يرتقي من كونه ملاحظة تجريبية معزولة إلى قانون تصميمي مؤكَّد تجريبياً بثلاث خوارزميات مستقلة: زمن الاستدلال في العنقود الموزع يتحدد بشكل شبه ثابت بحمل الشبكة (Network-Bound)، بينما يتحدد في البيئة المركزية بسرعة المعالج (Compute-Bound)، وكلما كان النموذج المستخدَم للتنبؤ أخف حسابياً، كانت الفجوة بين البيئتين أوسع لصالح المعالجة المركزية.

6. تحليل مقاييس الدقة: أول انحراف طفيف عن التكافؤ المطلق

خلافاً لتجربتَي RF وGBT، حيث ظل الفارق في مقاييس الدقة بين البيئتين ضئيلاً بشكل شبه متطابق، تُظهر تجربة MLP أول فارق ملحوظ نسبياً لصالح البيئة الموزعة، وتحديداً في دقة المرحلة الثانية (Multiclass) بفارق 1.03% (95.96% في Spark مقابل 94.93% في Sklearn)، وهو ما انعكس على الدقة الكلية الموحّدة (97.98% مقابل 97.77%).

ومع أن هذا الفارق يبقى ضمن هامش يمكن اعتباره متقارباً عملياً، إلا أن اتساقه في الاتجاه (لصالح Spark في كل من الدقة وF1-Score) يستدعي تفسيراً علمياً بدلاً من إرجاعه إلى ضجيج إحصائي عشوائي. ويُعزى هذا التميز الطفيف إلى طبيعة حساب الأوزان العصبية في بيئة موزعة، حيث يتم تحديث الأوزان استناداً إلى حزم بيانات (Batches/Partitions) موزعة عبر عقد مختلفة تحمل توزيعات فرعية متباينة قليلاً للبيانات. وهذا التنوع في مصادر التدرجات المجمَّعة عند كل خطوة تحديث يمنح النموذج الموزع مقاومة أعلى نسبياً للوقوع في نقاط استقرار محلية فرعية ( Local Minima) خلال عملية التحسين، مقارنة بالتدريب المركزي التتابعي الذي يُعالج البيانات بترتيب أكثر انتظاماً وقد يكون أكثر عرضة للتقارب المبكر نحو حل دون أمثل (Suboptimal Convergence)، خصوصاً في ظل عدد تكرارات محدود نسبياً (maxIter = 60).

هذه النتيجة تُبرز فائدة ثانوية للتوزيع الحوسبي لم تظهر بنفس الوضوح في التجربتين السابقتين: فبالإضافة إلى تسريع التدريب (في حالات معينة) وضمان الاستقرار الرقمي (كما في GBT)، قد يُسهم التوزيع أيضاً في تحسين جودة التعميم الإحصائي للنماذج التكرارية الحساسة لترتيب معالجة البيانات، وإن كان هذا الأثر في حالة MLP طفيفاً بما لا يبرر وحده تبني بنية موزعة مكلفة زمنياً في التدريب.

7. الاستنتاج المقارن: اكتمال الصورة عبر الخوارزميات الثلاث

تضيف تجربة MLP بُعداً ضرورياً لاستكمال الصورة المنهجية الشاملة لهذا الفصل من الدراسة، إذ تكشف أن الافتراض الشائع بأن Spark أسرع دوماً في التدريب هو افتراض غير دقيق حتى عند التعامل مع بيانات ضخمة (فوق 5.3 مليون سجل)، متى كانت الخوارزمية تتطلب تزامناً متكرراً بين العقد يفوق الفائدة المكتسبة من توزيع عملياتها الحسابية الأساسية الخفيفة نسبياً.

وبجمع نتائج التجارب الثلاث معاً، يتشكل نمط منهجي متكامل يمكن تلخيصه في ثلاث قواعد تصميمية مترابطة:

أولاً، جدوى التوزيع في مرحلة التدريب تتناسب طردياً مع حجم البيانات وتعقيد العملية الحسابية داخل الوحدة الأساسية للتدريب (شجرة قرار مفردة أو تحديث كامل لأوزان شبكة)، وتتراوح من انعدام الجدوى أو انعكاسها (RF على بيانات صغيرة، وMLP بشكل عام) إلى تفوق حاسم يصل لعشرة أضعاف (GBT).

ثانياً، البيئة الموزعة قد تتجاوز كونها أداة تسريع لتصبح ضامناً للاستقرار الرقمي للتدريب ذاته، كما ثبت حصراً في حالة GBT.

ثالثاً، وبثبات مطلق عبر الخوارزميات الثلاث دون استثناء واحد، تتفوق البيئة المركزية بفارق كبير في مرحلة الاستدلال اللحظي، وتتسع هذه الفجوة كلما خفّت الكثافة الحسابية للنموذج المستخدَم.

- **التجربة الرابعة: التوسع إلى الحجم الكامل للبيانات (Full-Scale Stress Test)**

1. الإطار المنهجي للتجربة

تختلف هذه التجربة جوهرياً عن التجارب الثلاث السابقة من حيث الهدف والمنهجية: فبينما سعت التجارب السابقة إلى مقارنة الأداء بين البيئتين على عينة موحّدة ثابتة (25% من CIC-IDS2018)، تهدف هذه التجربة إلى اختبار حدود التوسع (Scalability Limits) لكل بيئة عند الانتقال إلى السعة القصوى الكاملة للبيانات — 16,141,612 سجلاً شبكياً (100% من كتلة البيانات) — باستخدام خوارزمية الغابة العشوائية بنفس المعاملات الفائقة المعتمَدة سابقاً (100 شجرة، عمق أقصى 20).

والغاية العلمية من هذه التجربة ليست فقط قياس "من هو الأسرع"، بل الإجابة عن سؤال أكثر جوهرية بالنسبة لمشروع أمني يُفترض أن يعمل على بيانات مرورية حقيقية ومتصاعدة الحجم باستمرار: هل تصمد كل بيئة أصلاً أمام هذا الحجم من البيانات، أم أن هناك عتبة تتجاوزها إحدى البيئتين لتفشل كلياً؟

3. جدول النتائج

| المقياس الأدائي والحسابي | بيئة Spark الموزعة (3 عُقد) | البيئة المركزية (Single-Node) | الاستنتاج الهندسي والعملي |
|---|---|---|---|
| **حجم البيانات المعالجة** | **16,141,612 سجل** (الكامل) | **16,141,612 سجل** (الكامل) | اختبار القدرة القصوى للنظام |
| **زمن تدريب المرحلة الأولى (Binary)** | 4791.02 ثانية (~79.8 دقيقة) | **فشل ذريع (CRASH)** ✸ | انهيار تام للعقدة المنفردة |
| **زمن تدريب المرحلة الثانية (Multiclass)** | 641.65 ثانية (~10.7 دقيقة) | لم يتم الوصول إليها ✕ | - |
| **إجمالي زمن تدريب النظام (Total Train)** | **5432.67 ثانية (~90.5 دقيقة)** | لfinal خروج عن الخدمة ● | **Spark يعالج ما تعجز عنه العقدة المفردة** |
| **إنتاجية الاستدلال ( Stage 1 Throughput)** | **25,271.1 سجل/ثانية** | 0 سجل/ثانية | تفوق بنيوي للنظام الموزع |
| **حالة الذاكرة الافتراضية والفيزيائية** | استقرار كامل عند تخصيص GB3 | **امتلاء مطلق للـ RAM واختناق الـ VM** | استهلاك عتادي غير محكوم (Scikit-Learn) |

3. ظاهرة النمو دون الخطي: الكفاءة الاقتصادية المتصاعدة لبنية Spark

تكشف الأرقام المسجَّلة لعنقود Spark عند الانتقال من عينة الـ25% (4.04 مليون سجل، كما في التجربة الأولى) إلى العينة الكاملة (16.14 مليون سجل) عن نتيجة بالغة الدلالة: فرغم أن حجم البيانات المدخَلة تضاعف بمقدار 4 أضعاف بالضبط، إلا أن زمن تدريب المرحلة الأولى لم يرتفع بنفس هذه النسبة إطلاقاً، بل زاد بمقدار 2.93 ضعف فقط (من 1,637.80 ثانية إلى 4,791.02 ثانية)، وتكرر النمط ذاته في المرحلة الثانية بزيادة قدرها 2.63 ضعف فقط. والأهم من ذلك، أن إنتاجية المعالجة (Throughput) في المرحلة الأولى ارتفعت فعلياً بنسبة 44.4% (من 17,496 إلى 25,271 سجل/ثانية) بدلاً من أن تنخفض كما قد يُتوقَّع عند زيادة الحمل.

يُعرَف هذا النمط الرياضي هندسياً بـ Sub-linear Time Complexity Scaling، وهو يعني أن زمن المعالجة ينمو بمعدل أبطأ من معدل نمو حجم البيانات ذاته. ويُفسَّر هذا السلوك بأن التكلفة الزمنية الثابتة للتنسيق الشبكي بين العقد ( Fixed Communication & Scheduling Overhead) — وهي التكلفة نفسها التي أثبتت التجارب السابقة أنها تُثقل كفة Spark عند التعامل مع أحجام بيانات صغيرة (كما في المرحلة الثانية من تجربة RF على عينة الـ25%، وكما في تجربة MLP بأكملها) — تُصبح نسبياً ضئيلة ومُهملة كلما تعاظمت كتلة البيانات المعالَجة فعلياً محلياً داخل كل Worker. بعبارة أخرى، فإن نفس "العبء الثابت" الذي كان يُبطئ Spark في التجارب السابقة على البيانات الصغيرة، يتحوّل هنا إلى نسبة متضائلة الأثر كلما زاد حجم البيانات، بينما يبقى الجزء القابل للتوازي (Parallelizable Workload) هو المهيمن على الزمن الكلي.

هذه النتيجة تُثبت تجريبياً مبدأً اقتصادياً هندسياً مهماً يُعرف بـ وفورات الحجم (Economies of Scale): كفاءة النظام الموزع لا تبقى ثابتة، بل تتحسّن فعلياً كلما كبر حجم البيانات المدارة — وهي خاصية جوهرية ومرغوبة تماماً لنظام كشف تسلل يُفترض أن يتعامل مع حجم بيانات مروري متصاعد باستمرار مع نمو الشبكة السحابية المحمية.

4. انهيار العقدة المفردة: من "أبطأ" إلى "غير قابلة للتشغيل إطلاقاً"

يُمثّل هذا المحور نقطة التحوّل الأهم عبر التجارب الأربع مجتمعة، إذ ينقل طبيعة المقارنة من سؤال "أيّهما أسرع؟" إلى سؤال أكثر حسماً: "أيّهما قادر على العمل من الأساس؟". فمحاولة إخضاع البيئة المركزية (Scikit-learn) لنفس الحجم الكامل من البيانات لم تُسفر عن تباطؤ كبير كما في التجارب السابقة، بل عن انهيار فوري وتام (Crash) لكل من المكتبة البرمجية والآلة الافتراضية المستضيفة لها.

ويُعزى هذا الانهيار إلى فرق معماري جوهري في إدارة البيانات بين البيئتين تحت غطاء نظام التشغيل: تعتمد مكتبة Scikit-learn على تحميل مصفوفة الخصائص الكاملة ذات الأبعاد الضخمة دفعة واحدة بالكامل داخل الذاكرة العشوائية الفيزيائية (In-Memory Array Representation)، دون أي آلية لتجزئة هذا الحمل أو معالجته على دفعات. ومع بدء خوارزمية البحث عن نقاط التقسيم المثلى لبناء الأشجار المئة، تضاعف استهلاك الذاكرة بشكل أسّي حتى استنفدت العملية سعة الـ16 جيجابايت المتاحة بالكامل على الآلة، مما استدعى تفعيل آلية الحماية التلقائية في نظام التشغيل (Out-Of-Memory Killer) لإنهاء العملية قسرياً حفاظاً على استقرار النواة (Kernel) ذاتها — وهو ما أدى في النهاية إلى خروج الآلة بالكامل عن الخدمة.

في المقابل، أكملت بيئة Spark المهمة كاملة خلال 90.5 دقيقة فقط، وبحصة ذاكرة مُقيَّدة عمداً بـ3 جيجابايت فقط لكل Executor — أي أقل بخمس مرات تقريباً من إجمالي الذاكرة التي استُنفدت بالكامل في البيئة المركزية. ويعود هذا الاستقرار العتادي إلى معمارية الـ RDD (Resilient Distributed Datasets) الذكية التي يعتمدها Spark، والتي تُقسِّم البيانات إلى أجزاء (Partitions) قابلة للمعالجة المستقلة، وتعتمد مبدأ الحوسبة الكسولة (Lazy Evaluation) الذي يؤجّل التنفيذ الفعلي حتى الحاجة الحقيقية إليه، مع آلية تفريغ تلقائي للبيانات الزائدة إلى القرص الصلب عند الاقتراب من حدود الذاكرة (Spilling to Disk) بدلاً من محاولة الاحتفاظ بكل شيء في الذاكرة العشوائية دفعة واحدة كما تفعل Sklearn.

5. الدلالة المنهجية: من "تفضيل الأداء" إلى "ضرورة بنيوية"

تحمل نتيجة هذه التجربة ثقلاً أكاديمياً يتجاوز بكثير التجارب الثلاث السابقة مجتمعة، للسبب التالي: في التجارب الأولى (RF، GBT، MLP)، كانت المفاضلة بين البيئتين دوماً مسألة كفاءة نسبية — قد تكون إحدى البيئتين أسرع من الأخرى بمقدار 2 أو 10 أو حتى 28 مرة، لكن كلتا البيئتين كانتا قادرتين على إنجاز المهمة في النهاية. أما في هذه التجربة، فإن المقارنة لم تعد بين "سريع" و"أسرع"، بل بين "يعمل" و**"لا يعمل إطلاقاً"**.

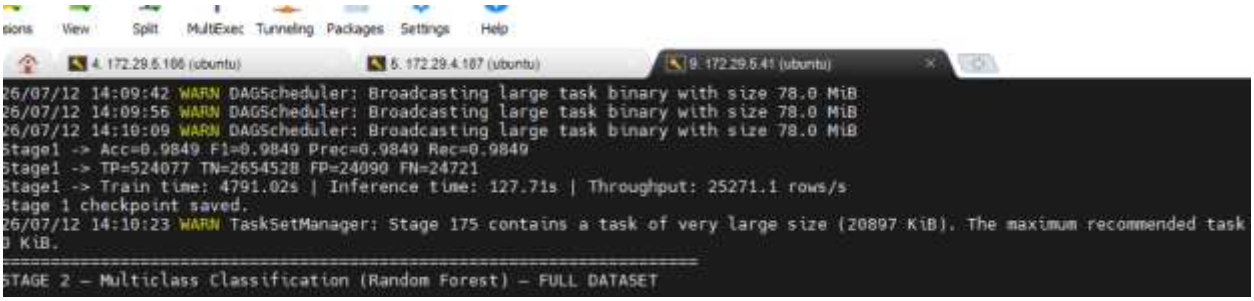
وهذا يُعيد صياغة السؤال البحثي الأساسي لهذا الفصل بأكمله: فبينما أثبتت التجارب الثلاث الأولى أن جدوى الحوسبة الموزعة مشروطة بعوامل عدة (حجم البيانات، طبيعة الخوارزمية، حاجة التزامن)، تُثبت هذه التجربة الرابعة أن هذا الشرط له عتبة حرجة (Critical Threshold): ما دون هذه العتبة، يبقى الاختيار بين البيئتين مسألة تحسين أداء (Performance Optimization)؛ أما عند تجاوزها — كما هو الحال حتماً في بيئة إنتاجية حقيقية لنظام كشف تسلل يراقب حركة مرورية ضخمة ومتصاعدة باستمرار — فإن غياب البنية الموزعة يتحوّل من عيب في الأداء إلى استحالة تشغيلية كاملة.

6. الاستنتاج الختامي: تكامل نتائج التجارب الأربعة

بجمع نتيجة هذه التجربة مع التجارب الثلاث السابقة، تكتمل الحجة العلمية الكاملة لهذا الفصل من الدراسة على النحو التالي:

أثبتت التجربة الأولى (RF على عينة 25%) أن فائدة التوزيع مشروطة بحجم البيانات ضمن نطاق معقول. أما هذه التجربة الرابعة، فتُضيف البُعد الأكثر حسماً على الإطلاق: عند بلوغ البيانات حجماً كافياً (وهو سيناريو حتمي وليس افتراضياً لنظام IDS حقيقي يعمل على مدار الساعة)، تنعدم أصلاً إمكانية الاستغناء عن البنية الموزعة، بصرف النظر عن أي اعتبارات أخرى تتعلق بتفضيل الأداء النسبي.

وبذلك، تُقدِّم التجارب الأربع مجتمعة إجابة متدرجة ومتكاملة عن سؤال جدوى اعتماد Apache Hadoop وApache Spark في مشروع Smart-IDPS: فالتوزيع ليس خياراً مثالياً في كل سيناريو (كما تُظهر حالات RF على بيانات صغيرة وMLP)، لكنه أيضاً ليس رفاهية تقنية اختيارية، بل ضرورة بنيوية حتمية عند التعامل مع الحجم الحقيقي للبيانات المرورية في بيئة سحابية واسعة النطاق — وهو بالضبط السياق التطبيقي الذي صُمِّم من أجله هذا المشروع أصلاً.




الشكل 20:نتائج تدريب المرحلة الاولى من النموذج المدرب على العنقود الموزع على كامل مجموعة البيانات

الشكل 21 : نتائج المرحلة الثانية من تدريب نموذج الذكاء الصنعي على العنقود الكامل على كامل مجموعة البيانات

سنعتمد Random Forest كخوارزمية أساسية لبناء النظام الهرمي ثنائي المرحلة في هذا المشروع.

من الجدير بالذكر أن هذا القرار لم يُتَّخذ بناءً على تفوّق حاد في مقاييس الدقة أو الزمن، إذ أظهرت النتائج أن الخوارزميات الثلاث حققت أداءً متقارباً جداً من حيث الـ Accuracy وF1-Score، وكذلك تقارباً ملحوظاً في زمن التدريب و خاصة Random Forest و GBT. إلا أن الفارق الحاسم ظهر بوضوح على مستوى الاستقرار الحسابي والملاءمة البنيوية للمعالجة الموزّعة، وهي معايير لا تقل أهمية عن الدقة والزمن في سياق نظام أمني يُفترض أن يعمل ضمن بيئة إنتاجية موزّعة

## 2.6- توازي البيانات مقابل توازي النموذج

أولاً — الخيار المعتمد عبر التجارب السابقة: توازي البيانات

من المهم التأكيد بوضوح أن استراتيجية التوزيع المعتمَدة بثبات عبر التجارب السابقة (RF، GBT، MLP) كانت توازي البيانات ( Data Parallelism) حصراً، وليس توازي النموذج (Model Parallelism). ولم يكن هذا الاختيار افتراضياً أو ناتجاً عن قصور في المعرفة بالبدائل المتاحة، بل قراراً منهجياً مبنياً على تقييم فني مباشر للبديل الآخر ورفضه لسببين موضوعيين:

السبب الأول — عدم الحاجة الفنية: أحجام النماذج الثلاثة المدرَّبة في هذا المشروع صغيرة جداً وتتّسع بسهولة تامة في ذاكرة عقدة حوسبية واحدة؛ فمئة شجرة قرار في RF أو GBT لا تتجاوز حجمها بضعة ميغابايتات إجمالاً، وشبكة MLP بطبقات لا تتعدى 50 عصبوناً لا تحتوي سوى آلاف قليلة من الأوزان القابلة للتدريب. توزيع نموذج بهذا الحجم الصغير فيزيائياً بين عقد متعددة لا يحل أي عائق حسابي حقيقي، بل يخلق عائقاً جديداً غير ضروري (Network Overhead) دون أي فائدة تقابله.

السبب الثاني — القيد التقني في البنية المستخدَمة: إطار Spark MLlib المعتمَد في هذا المشروع لا يوفّر أصلاً آلية توازي النموذج للخوارزميات الثلاث المستخدَمة؛ فهذه الآلية حصرية بأطر عمل متخصصة في التعلّم العميق الموزع للنماذج الضخمة جداً (كنماذج اللغة الكبيرة أو الشبكات التلافيفية العميقة جداً) والتي لا تتّسع أصلاً في ذاكرة معالج واحد — وهو سيناريو لا ينطبق إطلاقاً على نماذج كشف التسلل المطوَّرة هنا.

وبناءً على هذا التقييم، فإن توازي البيانات لم يكن مجرد "الخيار الوحيد المتاح"، بل الخيار المنهجي الصحيح فعلياً لطبيعة النماذج وحجمها في سياق هذا المشروع تحديداً.

ثانياً — الأثر العملي لهذا القرار على نتائج التجارب

هذا التمييز بين النوعين من التوازي هو بالضبط ما يُفسِّر النمط الثابت الذي رُصد عبر التجارب الثلاث مجتمعة: ففي مرحلة التدريب، حيث النموذج قيد البناء فعلياً وتتوزع البيانات الضخمة (تصل إلى ملايين السجلات) على العقد لتُحسَب الإحصائيات والتدرجات الجزئية محلياً، تحقق توازي البيانات فائدته الحقيقية — وبلغت هذه الفائدة ذروتها في تجربة GBT بتسريع بلغ 10.07 مرة. أما في مرحلة الاستدلال، حيث لا يوجد أصلاً حجم بيانات ضخم يستدعي التوزيع (بل دفعات صغيرة نسبياً من السجلات الجديدة)، فإن التكلفة الثابتة لتنسيق العنقود (Cluster Coordination Overhead) — وهي تكلفة ملازمة لأي توازي بيانات بصرف النظر عن حجم المهمة — أصبحت هي المهيمنة على الزمن الكلي، وهو ما فسّر تفوق البيئة المركزية في هذه المرحلة عبر الخوارزميات الثلاث دون استثناء.

ثالثاً — من التجربة إلى التصميم: تكرار النموذج (Model Replication) في مرحلة النشر

انطلاقاً من هذا الفهم الدقيق للفرق بين توازي البيانات وتوازي النموذج، اتُّخذ قرار معماري واعٍ عند الانتقال من مرحلة التجريب إلى مرحلة تشغيل النظام الفعلي: فبدلاً من الإبقاء على النموذج داخل بنية Spark الموزعة أثناء الاستدلال اللحظي (وهو ما أثبتت التجارب الثلاث أنه يُبطئ الأداء دون مبرر)، أو الاكتفاء بنسخة واحدة مركزية للنموذج على عقدة واحدة (وهو ما يُحوّل تلك العقدة إلى نقطة اختناق ونقطة فشل وحيدة — Single Point of Failure)، اعتُمد أسلوب ثالث هو تكرار النموذج (Model Replication): نُشرت نسخة كاملة ومطابقة من النموذج النهائي المصدَّر

(Exported Model) على كل من العقدتين k2 وk3 بشكل مستقل، بحيث تحتفظ كل عقدة بنسخة ذاتية الاكتفاء ( Self-Contained Copy) من النموذج جاهزة للاستدلال محلياً دون الحاجة لأي تنسيق مركزي عبر Spark Driver.

رابعاً — العائد المباشر: الإنتاجية وليس فقط التسريع

الفائدة الجوهرية من هذا القرار المعماري ليست تسريع عملية استدلال واحدة (فتلك أصلاً سريعة جداً على عقدة واحدة كما أثبتت التجارب)، بل رفع الإنتاجية الكلية للنظام (System-Wide Throughput) في بيئة التشغيل الحقيقية متعددة السيرفرات الافتراضية. فبدلاً من أن تعتمد جميع السيرفرات الافتراضية الثلاث (VM-01، VM-02، VM-03) على نسخة مركزية واحدة من النموذج تُعالِج طلبات الاستدلال الواردة من الجميع بشكل تسلسلي عبر نقطة وصول واحدة، أصبح بإمكان كل عقدة (k2 وk3) معالجة تدفقات السجلات الواردة إليها محلياً وبالتوازي التام مع العقدة الأخرى — أي أن النظام أصبح قادراً على معالجة عدة تدفقات شبكية بشكل متزامن فعلي (True Concurrency) بدلاً من الاصطفاف خلف نسخة واحدة من النموذج.

بعبارة أدق: التكرار هنا يخدم هدفاً مختلفاً تماماً عن التوزيع. فبينما كان الهدف من توزيع البيانات أثناء التدريب هو تسريع بناء نموذج واحد ضخم، فإن الهدف من تكرار النموذج أثناء الاستدلال هو تمكين معالجة متوازية لعدة طلبات مستقلة في آنٍ واحد، وهو ما يرفع سعة النظام الاستيعابية (System Capacity) لمواجهة أحمال حركة شبكية متزايدة من مصادر متعددة — وهو بالضبط السيناريو الواقعي لنظام كشف تسلل يراقب عدة سيرفرات في وقت واحد، لا سيرفراً واحداً معزولاً.

الخلاصة المترابطة

يُقدّم هذا القرار حلاً هندسياً مُحكماً يُوفِّق بين نتيجتين متعارضتين ظاهرياً استُخلصتا من التجارب الثلاث: الحاجة إلى الاستفادة من إمكانات العنقود الموزع من جهة، وحقيقة أن الاستدلال المركزي أسرع بكثير من الاستدلال عبر Spark من جهة أخرى. فبدلاً من محاولة "توزيع" عملية استدلال خفيفة أصلاً لا تحتاج توزيعاً (كما أثبتت التجارب الثلاث)، اختير أسلوب تكرار الحوسبة اللامركزية (Decentralized Replicated Inference) الذي يحصل على أفضل ما في العالمين: سرعة الاستدلال المحلي المباشر على كل عقدة (كما في البيئة المركزية)، مع القدرة على التوسع الأفقي ومعالجة أحمال متعددة بالتوازي (وهي الفائدة الأصلية المرجوة من امتلاك عنقود متعدد العقد أصلاً) — وهو ما يُترجم عملياً إلى ارتفاع مباشر وملموس في إنتاجية النظام الكلية دون التضحية بزمن استجابة كل طلب استدلال منفرد.

# الفصل السابع

# تنفيذ النظام

*نقوم في هذا الفصل الإجراءات التطبيقية لبناء و نشر نظام كشف التسلل الموزع طبقة تلو الآخرى.*

## 1.7- بيئة التنفيذ

### 1.1.7- مقدمة

بعد استكمال مرحلتي التصميم النظري وتدريب النموذج، ينتقل هذا الفصل إلى الجانب التطبيقي للمشروع، موضّحاً كيفية بناء النظام طبقة تلو الأخرى، بدءاً من البنية التحتية التي تستضيف مكوّناته، مروراً بمسار تدفّق البيانات عبر مراحله المختلفة، وصولاً إلى طبقة التحليل الذكي التي تمثّل جوهر النظام. وقد تمّ الالتزام الكامل بالتصميم المعماري الموزّع الذي تم تحديده في الفصل السابق، بحيث يعكس كل مكوّن مُنفَّذ مبرراً هندسياً واضحاً وليس مجرد اختيار تقني عشوائي.

### 2.1.7- البنية التحتية الموزعة للعنقود

لبناء نظام أمني قادر على معالجة البيانات الضخمة ومواجهة ظروف التشغيل الحقيقية، تم إنشاء عنقود موزّع (Distributed Cluster) يتكون من ثلاث عقد فيزيائية/افتراضية متصلة عبر شبكة محلية عالية السرعة، وتتوزع على النحو التالي:

- العقدة الرئيسية (k1 - Master Node): تُستضيف الخدمات المركزية مثل Spark Master، وHadoop NameNode، وZookeeper، ومحرك البث Kafka، بالإضافة إلى واجهة التحكم والربط.
- عقدتا المعالجة والتخزين (k2, k3 - Worker Nodes): تُستضيفان حاويات العمل الموزعة مثل Spark Workers و Hadoop DataNodes، حيث تتولى هذه العقد الحوسبة التوازية الفعلية وتخزين الكتل الموزعة.

ترتبط العقد الثلاث عبر عناوين IP ثابتة (Static IP Addresses) ضمن نطاق محلي خاص (10.0.0.x)، مع ضبط إعدادات كشف الخدمات عبر جدول المضيفين (extra_hosts) لضمان تواصل الحاويات عبر العقد المختلفة بذات كفاءة وسرعة التواصل المحلي.

### 3.1.7 مبدأ الحوسبة بالحاويات (Containerization)

اعتمد تنفيذ النظام على مبدأ الحوسبة بالحاويات (Containerization) كوحدة أساسية للعزل والتشغيل، بدلاً من التثبيت التقليدي المباشر لكل خدمة على نظام التشغيل المضيف. ويقوم هذا المبدأ على تغليف كل مكوّن من مكوّنات النظام ضمن حاوية (Container) مستقلة، تمتلك بيئتها التشغيلية ومكتباتها الخاصة، دون أن تتعارض مع باقي المكوّنات أو مع البيئة المضيفة.

استند اختيار هذا المبدأ إلى ثلاثة اعتبارات هندسية رئيسية:

- العزل الموارد والأخطاء (Isolation): تعمل كل خدمة أمنية أو حوسبية في بيئة معزولة تماماً، بحيث لا يؤدي تعطّل أو إعادة تشغيل حاوية واحدة (مثل إعادة تشغيل أحد الـ DataNode على k2) إلى التأثير على باقي الخدمات العاملة في العنقود.
- قابلية إعادة الإنتاج والتوثيق (Reproducibility): تضمن الحاويات تماثل البيئات التشغيلية عبر جميع عقد العنقود (k1, k2, k3) بفضل بنائها من صور موحدة المحدد الإصدار (Docker Images)، مما يمنح التوثيق الأكاديمي موثوقية عالية ويتيح إعادة تكرار التجربة والنتائج بنفس الشروط.
- كفاءة استغلال الموارد العتادية: على عكس الأجهزة الافتراضية التقليدية (Virtual Machines) التي تتطلب نظام تشغيل كاملاً ومستقلاً لكل خدمة، تشترك الحاويات في نواة نظام التشغيل المضيف للنود، مما يقلل بشكل حاد من استهلاك الذاكرة (RAM) والمعالج (CPU)، ويسمح بتشغيل خدمات متعددة بكفاءة ضمن الحد الأدنى المتاح للموارد.

### 4.1.7 التنسيق وإدارة الخدمات عبر Docker Compose

في حين تمثل الحاوية وحدة العزل الأساسية، فإن إدارة الحاويات الموزعة عبر العقد الثلاث يتطلب أداة تنسيق (Orchestration) تُعرّف دورة حياة كل خدمة وعلاقتها بالخدمات الأخرى. لهذا الغرض استُخدمت أداة Docker Compose على مستوى كل عقدة، مما مكّن من:

- البنية التحتية ككود (Infrastructure as Code - IaC): تعريف ملفات تصريحية (docker-compose.yml) تصف الخدمات، الموانئ المشهورة، وحدود الموارد المخصصة (Memory & CPU Limits) لكل حاوية، مما جعل البنية التحتية موثقة تماماً وقابلة للتتبع عبر أنظمة التحكم بالإصدارات (Git).
- إدارة الاعتماديات التسلسلية (depends_on & Healthchecks): ضمان تشغيل الخدمات بترتيب منطقي محدد؛ على سبيل المثال، عدم تشغيل حاوية Kafka إلا بعد تأكيد جهوزية واكتمال إقلاع حاوية Zookeeper على العقدة الرئيسية.
- الربط الشبكي الهجين (Hybrid Networking): تعيين شبكة داخلية مشتركة (ids-net) للحاويات الموجودة على نفس العقدة لتتواصل عبر أسماء الخدمات (Service Discovery)، مدعومة بإعدادات extra_hosts التي تترجم أسماء الخدمات على العقد الأخرى إلى عناوين الـ IP الخاصة بالعنقود (10.0.0.15, 10.0.0.124, 10.0.0.18).
- تبسيط دورة التشغيل الصيانة: إدارة بيئة العمل الموزعة بالكامل عبر أوامر موحدة، مما أتاح إمكانية تشغيل أو إيقاف أو إعادة بناء الطبقات التشغيلية بسهولة وسرعة خلال مراحل الاختبار والضبط.

## 2.7- طبقات النظام

### 1.2.7 – طبقة التقاط و تجميع السجلات الشبكية

تمثّل هذه الطبقة نقطة الانطلاق والمدخل الأساسي لتدفق البيانات عبر أوردة النظام بأكمله. وتتلخص مسؤوليتها الهندسية في تحويل الحركة الشبكية الخام (Raw Network Traffic) إلى سجلات ذات ميزات مهيكلة (Structured Features)، ثم شحنها بشكل لحظي مستمر إلى قناة البث الموزع.

حرصاً على محاكاة واقع بيئات التشغيل الحقيقية وفي نفس الوقت احترام حدود العنقود الموزّع، تتوزع الحاويات المشكّلة لهذه الطبقة عبر عقد العنقود (وتحديداً على العقدة الفرعية k3 / vm-server-3)، وتتواصل مع الخدمات الموزعة المركزية في العقدة الرئيسية k1 (مثل NameNode وKafka) عبر بروتوكولات شبكية مُعرّفة باستخدام إعدادات العناوين الداخلية (extra_hosts) لضمان الفصل الوظيفي والتطابق الشكلي.

تتكون هذه الطبقة من ثلاث حاويات أساسية تعمل وفق خط إنتاج تسلسلي (Data Pipeline):

1. حاوية التقاط وتحويل الحركة (pcap-watcher)

تعمل هذه الحاوية كـ "مراقب آلي" للمجلد الإدخالي المستمر (/input). بمجرد استلام أي ملف التقاط شبكي خام بصيغة (.pcap) ناتج عن أجهزة المراقبة أو محاكاة الهجمات، تقوم الحاوية فوراً بتشغيل أداة التحليل الشبكي CICFlowMeter

التي تعمل على استخراج نفس الميزات الشبكية المعتمدة .

```
pcap-watcher:
  build: ./pcap-watcher
  container_name: pcap-watcher
  volumes:
    - ./pcaps_input:/input
    - ./source_attacks:/source_attacks
  networks: [ids-net]
  mem_limit: 512m
  restart: unless-stopped
```

2. حاوية التقطير والتأطير الزمني (dripper)

الوظيفة العملية: تعتمد هذه الحاوية على برمجية خاصة (traffic_dripper.py) لتقوم بقراءة ملفات CSV المستخرجة من الحاوية السابقة، ومن ثم إفراز سجلاتها سطرًا بسطر (Row-by-Row Streaming) بنمط زمني مُتحكّم به وإيداعها في مجلد الإخراج (/output).

كما تتولى هذه الحاوية التواصل مع العقدة الرئيسية لـ HDFS عبر الشبكة (namenode:10.0.0.15) لأرشفة الملفات المعالجة، مع اعتماد آلية التعلّيم بملفات الأثر (dripping markers.) لمنع تكرار معالجة البيانات نفسها.

```yaml
dripper:
  build: ./dripper
  container_name: dripper
  volumes:
    - ./source_attacks:/source_attacks
    - ./output:/output
  extra_hosts:
    - "namenode:10.0.0.15"
  depends_on:
    - pcap-watcher
  networks: [ids-net]
  mem_limit: 512m
  restart: unless-stopped
```

3. حاوية النقل والشحن اللحظي (filebeat / filebeat-vm3)

الوظيفة العملية: تُبنى هذه الحاوية انطلاقاً من بيئة السيرفر (./vm-server) المخصصة للمراقبة، وتعمل كـ Log Shipper خفيف الوزن يستهدف مجلد الإخراج (/output). تقوم بـ "تعقب أطراف السجلات" (Log Tailing) فور كتابتها، وتقوم بتأطيرها وإرسالها عبر الشبكة مباشرة إلى موزع الرسائل Kafka على العقدة الرئيسية (kafka:10.0.0.15).

تم ربط مجلد الإخراج بالحاوية بصلاحية القراءة فقط(:ro)، مما يحقق مبدأ العزل الصارم ؛ إذ لا تملك حاوية النقل أي قدرة على تعديل البيانات الأصلية أو تدميرها.

```yaml
# FILEBEAT
# ==========================================================
filebeat:
  build: ./vm-server
  container_name: filebeat-vm3
  hostname: vm-server-3
  volumes:
    - /home/ubuntu/ids-project/output:/output:ro
  extra_hosts:
    - "kafka:10.0.0.15"
  networks: [ids-net]
  mem_limit: 512m
  restart: unless-stopped
```

ميزات :

- الأمان والعزل: حُرِص على ربط مجلد الإخراج بالحاوية بصلاحية القراءة فقط (:ro)، مما يحقق مبدأ العزل الصارم؛ إذ لا تملك حاوية النقل أي قدرة على تعديل البيانات الأصلية أو تدميرها.

- الفصل الهيكلي (Decoupling): الفصل بين منطق توليد/معالجة الميزات (المستضاف في pcap-watcher وdripper) وبين منطق النقل والبث (المستضاف في filebeat) يمنح البنية التحتية مرونة عالية لإعادة توجيه البيانات مستقبلاً لجهات أخرى دون المساس بمنطق العمل الأساسي.

### 2.2.7 – طبقة النقل و البث اللحظي

تشكّل هذه الطبقة العمود الفقري لتدفّق البيانات داخل النظام، وتتكوّن من حاويتين متكاملتين وظيفياً:

Kafka هو المكوّن المسؤول عن استقبال السجلات القادمة من طبقة تجميع البيانات وإتاحتها فوراً للاستهلاك من قِبل طبقة المعالجة (Spark)، وذلك عبر آلية النشر والاشتراك (Publish-Subscribe)، حيث تُنشَر السجلات على قناة (Topic) محدَّدة، ويستهلكها محرك Spark لحظة وصولها دون الحاجة لتخزينها وسيطاً أو انتظار تجمّعها في دفعات.

Zookeeper هو خدمة تنسيق موزّعة (Distributed Coordination Service) يعتمد عليها Kafka لإدارة حالته الداخلية، ولا يقتصر دورها على ترجمة الأسماء كما قد يُظَن للوهلة الأولى، بل تتولّى تسجيل حالة وسيط (Broker) Kafka وتتبّع جهوزيته، وتنسيق عملية اختيار "المسؤول" (Leader) عن كل قسم من أقسام البيانات (Partitions) في حال تعدد الوسطاء، إضافة إلى الاحتفاظ بإعدادات القنوات (Topics) بشكل مركزي. وبذلك يضمن Zookeeper استقرار عملية البث وتماسكها، خصوصاً في السيناريوهات التي يتوسّع فيها النظام مستقبلاً ليشمل أكثر من وسيط Kafka واحد، وهو ما يخدم متطلب قابلية التوسع (Scalability) الذي حُدِّد ضمن المتطلبات غير الوظيفية للنظام.

### 3.2.7 – طبقة المعالجة و التحليل اللحظي

تمثّل هذه الطبقة العقل المفكّر والمحلِّل للنظام؛ حيث تُطبَّق نماذج الذكاء الاصطناعي الهرمية على السجلات الشبكية الحية القادمة من طبقة البث اللحظي (Kafka)، وتحويلها من بيانات خام غير مصنفة إلى قرارات أمنية ودقيقة.

لم يكن اختيار محرك Apache Spark Streaming واستضافته على عنقود موزّع خياراً تقنياً مجرداً، بل جاء معالجةً هندسيةً حاسمة لثلاث مشكلات جوهرية في أنظمة كشف التسلل الشبكي الحديثة:

- معالجة معضلة "اختناق المعالجة المباشرة" (Throughput Bottleneck):

في الشبكات الحديثة، تصل السجلات الشبكية بمعدلات تتجاوز آلاف التدفقات في الثانية. إن استخدام سيرفر أحادي (Single Node) لعمليات استدلال الذكاء الاصطناعي (AI Inference) يُؤدي فوراً إلى تراكم الرسائل في الذاكرة وتأخر زمن الاستجابة (Latency). عبر استخدام Spark، يتم تقسيم دفق البيانات الوارد من Kafka إلى دفعات ميكروية (Micro-batches كل 15 ثانية)، وتوزيع معالجتها بالتوازي على العقد العاملة.

- توزيع حمل النماذج الهرمية (Hierarchical AI Inference Distribution):

يستخدم النظام نموذجاً هرمياً يتطلب تطبيق مرحلتين من التنبؤ (Stage 1: Logistic Regression ثم Stage 2: Random Forest). تحميل هذه النماذج وإجراء العمليات الحسابية المتجهة (Vector Transformations) لكل تدفق شبكي يُعدّ مكلفاً حوسبياً ( CPU/RAM

Intensive). بتوزيع هذه النماذج على العقدتين العاملتين (k2 وk3)، تنفذ العمليات الحسابية بالتوازي على أجزاء مختلفة من نفس الدفعة، مما يحافظ على زمن استجابة لحظي.

- تجنب نقطة الفشل الوحيدة والتوسع الأفقي (Horizontal Scalability):

تضمن بنية العنقود استمرارية العمل؛ حيث تُستضاف حاوية المنسق الرئيسي (spark-master) على العقدة الأولى (k1) لتتولى جدولة المهام ( Job Scheduling) دون إرهاقها بالمعالجة الحسابية، بينما ينحصر التنفيذ الفعلي على العقد العاملة (spark-worker1 على k2: 10.0.0.124 وspark-worker2 على k3: 10.0.0.18).

تُدار هذه الطبقة عبر البرمجية الموزعة ids_streaming.py والمشغَّلة من العقدة الرئيسية، حيث تتبع خط إنتاج برمجي دقيق

1. استهلاك الدفق وبناء المخطط البنيوي (Stream Parsing)

2. الاستدلال الهرمي ثنائي المرحلة (Hierarchical Inference)

3. معالجة معضلة "فيضان التنبيهات" والتجميع اللحظي (Alert Flood Mitigation)

عند وقوع هجوم شبكي (مثل DDoS)، قد يُرسِل المهاجم مئات التدفقات في الدفعة الواحدة (15 ثانية)، مما قد يغرق لوحة التحكم بآلاف التنبيهات المتكررة.

لمعالجة هذه المشكلة، يُطبق السكريبت عملية تجميع لحظي (In-Memory Aggregation) داخل كل دفعة ميكروية:

تُجمّع السجلات المشبوهة بناءً على المزيج: (src_ip, attack_type).

تُحسب إجمالي التدفقات القادمة من هذا المصدر عبر المتغير occurrence_count.

يُبنَى تنبيه موحّد واحد يحتوي على ملخص الهجوم، وشدته (HIGH للهجمات الشديدة كـ DDoS وMEDIUM لغيرها)، وعينة من الميزات (sample_features).

يُرسل التنبيه المجمع عبر طلب HTTP POST بمهلة زمنية محدودة (timeout=3s) إلى الخدمة الخلفية (FastAPI) المستضافة على العقدة الثانية

4. الأرشفة الدقيقة الموزعة في HDFS لغايات إعادة التدريب (Granular Archiving)

على العكس من التنبيهات المرسلة للوحة التحكم (والتي تم تجميعها)، يتطلب إعادة تدريب النماذج مستقبلاً وجود البيانات على مستوى التدفق المفرد بكل تفاصيله.

لذا، يُعيد السكريبت دمج نتائج الحركة الطبيعية والحركة المهاجمة قبل التجميع، ويقوم بكتابتها بصيغة Parquet المحسّنة والمضغطَة إلى نظام الملفات الموزع HDFS على المسار:

تتيح هذه الأرشفة الموزعة مقسمةً بحسب التاريخ (Date Partitioning) إمكانية ربط هذه السجلات لاحقاً مع أحكام المحلل الأمني المخزنة في قاعدة البيانات (PostgreSQL) لبناء مجموعات تدريبية جديدة عالية الجودة بكفاءة وسرعة.

### 4.2.7 – طبقة الخدمة التخزين الدائم و الموزع

تتولّى هذه الطبقة الأرشفة الدائمة لسجلات الحركة الشبكية بعد مرورها بطبقتي التجميع والبث، وتعتمد على نظام Hadoop HDFS الموزّع فعلياً عبر عقد العنقود الثلاث، بما يحقّق مبدأ التخزين الموزّع المتسامح مع الأعطال الذي بُني عليه اختيار Hadoop في التصميم المعماري للنظام.

تتوزّع مكوّنات هذه الطبقة على النحو التالي: تُستضاف حاوية namenode على العقدة الأولى (k1)، وتتولّى إدارة بنية الملفات الموزّعة بالكامل (Namespace Management)، والاحتفاظ بخريطة توزيع كتل البيانات (Block Mapping) عبر عقد التخزين، دون أن تُخزّن هي نفسها أي بيانات فعلية. أما تخزين البيانات الفعلي فيتم عبر حاويتي DataNode المستضافتين على عقدتين منفصلتين: الأولى (datanode) على العقدة الثانية (k2)، والثانية (datanode2) على العقدة الثالثة (k3)، بحيث يُخزَّن كل جزء من السجلات على أكثر من عقدة فعلية مستقلة، لا على حاويات متعددة ضمن العقدة نفسها، وهو ما يجسّد بمعناه الحقيقي مفهوم "التوزيع" لا مجرد "التكرار المحلي".

ويظهر هذا التوزيع الفعلي بوضوح في إعداد extra_hosts المضاف لكل من حاويتي DataNode، حيث تُربَط كل منهما بعنوان IP الداخلي الفعلي لعقدة NameNode (namenode:10.0.0.15) بدلاً من الاعتماد على تحليل الاسم عبر شبكة Docker المحلية، وذلك تحديداً لأن NameNode وDataNode في هذه الحالة لا يقعان على العقدة نفسها، فلا يمكن الاعتماد على آلية اكتشاف الخدمات الداخلية ( Service Discovery) المحدودة النطاق بعقدة واحدة، بل يتوجّب التصريح الصريح بعنوان العقدة المضيفة عبر الشبكة الداخلية للعنقود ( Internal Cluster Network).

يحقّق هذا التوزيع فائدتين جوهريتين: الأولى هي التسامح مع الأعطال، إذ يستمر النظام بالعمل حتى في حال تعطّل إحدى عقدتي DataNode بفضل آلية النسخ الاحتياطي (Replication) التي يوفّرها HDFS تلقائياً بين العقد المتاحة. والثانية هي قابلية التوسع الأفقي، حيث يمكن إضافة عقد تخزين جديدة على عقد إضافية بالعنقود دون الحاجة لإيقاف النظام أو إعادة هيكلة البيانات الموجودة مسبقاً، بما يخدم متطلب قابلية التوسع (NFR-01) المحدَّد ضمن المتطلبات غير الوظيفية.

### 5.2.7 – طبقة الخدمة الخلفية و التحكم

تمثّل هذه الطبقة، المستضافة على العقدة الثانية (k2)، حلقة الوصل بين نتائج التحليل الصادرة عن طبقة المعالجة الذكية وبين واجهة المستخدم النهائية، وتتولّى مسؤوليات متعددة تتجاوز مجرد نقل البيانات، لتشمل الاستقبال والتجميع، والمصادقة، والتخزين الدائم للأحداث، وإدارة دورة حياة النموذج نفسه. بُنيت هذه الطبقة باستخدام إطار FastAPI، ويمكن تفصيل وظائفها الجوهرية كما يلي:

أولاً — استقبال التنبيهات ومنطق الدمج (Alert Ingestion with Upsert Logic)

يستقبل نقطة النهاية /api/alerts التنبيهات المجمَّعة الواردة من طبقة Spark، إلا أن معالجتها لا تقتصر على الإدراج المباشر في قاعدة البيانات، بل تطبّق منطق دمج زمني (Upsert Logic): إذا ورد تنبيه جديد من العنوان نفسه ونوع الهجوم نفسه خلال نافذة زمنية مقدارها 60 ثانية من آخر تنبيه مطابق ما زال في حالة "قيد المراجعة" (pending)، يُحدَّث السجل الموجود برفع عداد التكرار (occurrence_count) بدلاً من إنشاء سجل جديد. وبهذا تُستكمَل معالجة فيضان التنبيهات التي بدأت على مستوى التجميع داخل Spark، لتُضاف طبقة حماية ثانية على مستوى قاعدة البيانات تمنع تكرار الحادثة الأمنية نفسها كسجلات منفصلة عبر دفعات Spark المتتالية.

ثانياً — البث اللحظي للتنبيهات (Real-Time Push via WebSocket)

عوضاً عن اعتماد لوحة التحكم على الاستعلام الدوري المتكرر (Polling) لمعرفة وجود تنبيهات جديدة، تُستخدم قناة WebSocket (/ws/alerts) تُدار عبر فئة ConnectionManager، بحيث تُبَث كل عملية إدراج أو تحديث لتنبيه فور وقوعها إلى جميع المتصفحات المتصلة بلوحة التحكم في آنٍ واحد، وهو ما يحقق فعلياً متطلب "الاستجابة في الوقت الفعلي" الذي حُدِّد كمتطلب غير وظيفي على مستوى النظام ككل، لا على مستوى محرك التحليل فقط.

ثالثاً — تجميع الموارد لقاعدة البيانات (Connection Pooling)

نظراً لأن نقطتي النهاية الخاصتين بالتنبيهات (GET وPOST /api/alerts) هما الأكثر عرضة للحمل المتكرر بحكم استقبالهما دفعات متتالية من Spark كل 15 ثانية، خُصِّص لهما تجمّع اتصالات مُدار (ThreadedConnectionPool، بحد أدنى 5 وحد أقصى 50 اتصالاً)، بدلاً من فتح وإغلاق اتصال جديد بقاعدة البيانات مع كل طلب، وهو ما يقلّل زمن استجابة هذين المسارين تحديداً تحت الحمل العالي، في حين تعتمد بقية نقاط النهاية الأقل تكراراً على اتصال مباشر تقليدي، بما يعكس قراراً هندسياً واعياً بتخصيص التحسين للمسار الحرج فقط دون تعقيد النظام بأكمله.

رابعاً — تقييم أداء النموذج وتشغيل إعادة التدريب (Model Performance and Retraining Trigger)

توفّر نقطة النهاية /api/model/performance تقييماً حياً لدقة النموذج، مبنياً على مقارنة تنبيهات النظام بأحكام المحلّل الأمني الفعلية (true_positive مقابل false_positive)، لا على دقة التدريب الثابتة وحدها. وبناءً على عتبتين محدَّدتين مسبقاً — انخفاض الدقة المؤكدة عن 85%، أو تراكم 200 تنبيهاً بانتظار المراجعة — يُوصي النظام تلقائياً بضرورة إعادة التدريب. وعند تفعيل ذلك عبر /api/retrain، تُنفَّذ عملية إعادة التدريب فعلياً على عقدة Spark Master (k1) عبر اتصال SSH بعيد يُطلق سكريبت retrain_model.py، مع تشغيل المهمة في الخلفية (Background Task) وتتبّع حالتها (idle/running/success/rejected) عبر نقطة نهاية منفصلة، بما يسمح للوحة التحكم بعرض تقدّم العملية دون حجب باقي وظائف الخادم.

خامساً — فحص سمعة العناوين مع التخزين المؤقت (IP Reputation Caching)

عند طلب تفاصيل أي تنبيه، يُستعلَم عن سمعة عنوان المصدر عبر خدمة AbuseIPDB الخارجية، مع تخزين النتيجة مؤقتاً في جدول ip_reputation_cache لمدة 24 ساعة، تجنّباً لاستهلاك حصة الاستعلامات الخارجية المحدودة عند تكرار الاستفسار عن العنوان نفسه، مع معالجة خاصة للعناوين الداخلية/المحاكاة التي لا تملك سمعة علنية أصلاً.

سادساً — المصادقة وإدارة الجلسات (Authentication)

تُطبَّق طبقة مصادقة مبسّطة قائمة على الجلسات (Session-based) عبر رمز عشوائي (Token) يُصدَر عند تسجيل الدخول ويُتحقَّق منه في العمليات الحساسة فقط (تحديث الأحكام، حظر/رفع حظر العناوين)، في حين تبقى نقاط النهاية الخاصة بالعرض والاستعلام مفتوحة دون مصادقة، بما يعكس تمييزاً واضحاً بين العمليات القرائية والعمليات المغيِّرة لحالة النظام.

### 5.2.7 – طبقة العرض

تمثّل هذه الطبقة، المستضافة كحاوية مستقلة ضمن العقدة الثانية (k2)، الواجهة النهائية التي يتفاعل من خلالها المحلّل الأمني مع النظام، وقد أُنجزت باستخدام مكتبة React. ولا تقتصر أهمية هذه الطبقة على العرض البصري للبيانات فحسب، بل تمتد لتشكّل حلقة الوصل الفعلية بين القرار الآلي الذي يصدره النموذج والقرار البشري النهائي الذي يتخذه المحلّل، بما يجعلها نقطة التقاء التحليل الآلي بالإشراف البشري في النظام ككل.

أولاً — تسجيل الدخول والتحكم بالوصول

يُشترَط على أي مستخدم المرور عبر شاشة تسجيل دخول قبل الوصول لأي من أقسام لوحة التحكم، بحيث تبقى الصفحات الوظيفية (النظرة العامة، التهديدات، أداء النموذج) محجوبة تماماً عن أي زائر غير مصرَّح له، ولا تُتاح إلا بعد تحقّق الخادم الخلفي من صحة بيانات الاعتماد المدخَلة.

ثانياً — التحديث اللحظي عبر WebSockets

يمثّل هذا الجانب أحد أهم القرارات التصميمية في طبقة العرض، ويستحق توضيحاً خاصاً لما يحمله من فرق جوهري عن أسلوب العرض التقليدي للوحات التحكم.

في النمط التقليدي، تعتمد لوحات التحكم على ما يُعرف بـ"الاستعلام الدوري" (Polling)، حيث يرسل المتصفح طلباً متكرراً للخادم كل فترة زمنية ثابتة ليسأل: "هل هناك بيانات جديدة؟" — وهذا الأسلوب يحمل عيبين جوهريين: الأول هو التأخير الحتمي بين لحظة وقوع الحدث الفعلي (كاكتشاف هجوم) ولحظة ظهوره على الشاشة، والذي يساوي في أفضل الحالات طول الفترة الزمنية بين استعلامين متتاليين؛ والثاني هو الحمل غير الضروري على الخادم نتيجة تكرار آلاف الطلبات الفارغة التي لا تحمل جديداً.

لتفادي هذا القصور، اعتمدت لوحة التحكم على تقنية WebSocket، وهي بروتوكول اتصال يفتح قناة دائمة ومستمرة بين المتصفح والخادم الخلفي، بدلاً من سلسلة طلبات منفصلة ومتكررة. وبمجرد فتح هذه القناة، يصبح بإمكان الخادم "دفع" (Push) أي بيانات جديدة فوراً إلى المتصفح فور توفّرها، دون أن ينتظر المتصفح أن يسأل عنها. وبهذا، فإن اللحظة التي يُصدر فيها محرك التحليل قراراً باكتشاف هجوم، تُبَث فيه فوراً إلى شاشة المحلّل عبر هذه القناة، محقِّقةً بذلك معنى "الوقت الفعلي" (Real-Time) في أدق تجلياته العملية، لا كمجرد شعار تسويقي.

ولضمان استمرارية هذا الاتصال حتى في حال انقطاعه لأي سبب طارئ (كإعادة تشغيل الخدمة الخلفية أو تذبذب الشبكة)، زُوِّدت لوحة التحكم بآلية إعادة اتصال تلقائية، مصحوبة بمؤشر بصري واضح يعكس للمحلّل حالة الاتصال في كل لحظة — متصل حياً، أو في طور إعادة المحاولة — بما يمنحه ثقة تامة بأن ما يراه على الشاشة يعكس فعلاً الحالة اللحظية للنظام.

ثالثاً — تنبيه المحلّل للتهديدات الحرجة

عند وصول تهديد ذي خطورة عالية عبر القناة اللحظية، لا تكتفي اللوحة بإضافته إلى قائمة التنبيهات، بل تُطلق إشعاراً منبثقاً مصحوباً بنغمة تنبيه سمعية، بهدف لفت انتباه المحلّل فوراً دون الحاجة لمراقبة الشاشة باستمرار، مع الاكتفاء بتنبيه صامت في القائمة للتهديدات الأقل خطورة، تفادياً لإرهاق المحلّل بإشعارات متكررة لا تستدعي تدخلاً عاجلاً.

رابعاً — فرز التنبيهات وإصدار الحكم البشري

توفّر اللوحة قسماً مخصَّصاً لمراجعة التنبيهات وفرزها حسب نوع الهجوم أو درجة الخطورة أو حالة المراجعة، مع إتاحة الاطلاع على الأدلة التفصيلية التي استند إليها النموذج في تصنيفه — من خصائص الحركة الشبكية الخام، إلى سمعة عنوان المصدر المستقاة من مصدر خارجي موثوق. وعلى أساس هذه الأدلة، يصدر المحلّل حكماً صريحاً بتأكيد الهجوم أو اعتباره إنذاراً كاذباً، وهذا الحكم البشري تحديداً هو ما يُغذّي لاحقاً عملية تقييم أداء النموذج، بحيث تتحوّل لوحة التحكم من أداة عرض سلبية إلى نقطة إدخال فعلية في حلقة التحسين المستمر للنظام.

خامساً — متابعة أداء النموذج وإدارة إعادة التدريب

تعرض اللوحة أيضاً صفحة مخصَّصة لمتابعة أداء النموذج، تجمع بين معلومات النموذج الأساسية (الخوارزمية، بيانات التدريب) ومقاييس حية مستمدة من أحكام المحلّلين الفعلية بعد النشر، إلى جانب إمكانية تفعيل عملية إعادة التدريب مباشرة من الواجهة عند الحاجة، ومتابعة تقدّمها لحظياً حتى اكتمالها.

## 3.7 التقنيات و الأدوات المستخدمة

### Apache Spark وأداة spark-submit

يُعدّ Apache Spark محرّك المعالجة الموزّعة الذي بُني عليه الجزء التحليلي بأكمله من النظام، سواء في مرحلة تجريب النموذج أو في مرحلة تشغيله الفعلي على البيانات الحية. وقد اعتُمدت أداة spark-submit، وهي الأداة القياسية المرفقة مع توزيعة Spark لإرسال برامج المعالجة (Jobs) إلى العنقود وتشغيلها، كنقطة الدخول الوحيدة لتنفيذ أي سكريبت Python يعتمد على مكتبة PySpark ضمن هذا المشروع.

استُخدمت هذه الأداة على مرحلتين متمايزتين من حيث الغرض:

في مرحلة التجريب، استُخدمت spark-submit لتشغيل سكريبتات تدريب النموذج وتقييمه بشكل تفاعلي، حيث أتاحت مراقبة نتائج كل تجربة فور اكتمالها — كمقاييس الدقة (Accuracy) ومقياس التوازن (F1-Score) ومصفوفة الالتباس (Confusion Matrix) — بما مكّن من مقارنة عدة نماذج (Logistic Regression وRandom Forest وDecision Tree) واختيار الأنسب منها بناءً على أدلة تجريبية موثّقة، لا على تفضيل نظري مجرّد.

في مرحلة التشغيل الإنتاجي، استُخدمت الأداة نفسها لإطلاق سكريبت ids_streaming.py كعملية مستمرة (Long-Running Job) على عقدة Spark Master، مع تمرير معاملات صريحة لضبط سلوك التشغيل الموزّع، من بينها عدد الأنوية المخصَّصة لكل تنفيذ (--total-executor-cores)، وحجم الذاكرة المخصَّص لكل منفِّذ (--executor-memory)، إضافة إلى إعدادات شبكية دقيقة كعنوان ومنفذ برنامج التشغيل (spark.driver.host وspark.driver.port)، وهي إعدادات كانت ضرورية لضمان تواصل سليم بين عقدة Master والعمّال الموزّعين على عقد العنقود المختلفة. وبهذا، لم تقتصر أداة spark-submit على كونها أداة تشغيل بسيطة، بل شكّلت الواجهة الموحَّدة التي تحكّمت بكامل السلوك الموزّع للنظام، بدءاً من التجربة الأولى وصولاً إلى التشغيل المستمر لخط أنابيب الكشف اللحظي.

### Docker وDocker Compose

Docker هو نظام مفتوح المصدر يُستخدم لإنشاء التطبيقات ونشرها وتشغيلها داخل حاويات (Containers)، وهي بيئة تشغيل معزولة تحتوي على كل ما تحتاجه الخدمة من كود ومكتبات وإعدادات تشغيل، بما يضمن عمل التطبيق بالشكل نفسه عبر مختلف البيئات دون اعتماد على خصائص الجهاز المضيف. ويقوم هذا النظام على مبدأ التجريد على مستوى نظام التشغيل (OS-level Virtualization)، وهو ما يجعله أخف بكثير من الأجهزة الافتراضية التقليدية (Virtual Machines)، ويتيح تشغيل عدد كبير من الحاويات على الجهاز نفسه دون استنزاف موارده.

اعتمد هذا المشروع على أربعة مفاهيم أساسية من مفاهيم Docker: الصورة (Image) التي تمثّل وصفاً ثابتاً وموثّقاً لكل خدمة، بما يشمل نظام التشغيل والتطبيق وكافة اعتمادياته؛ الحاوية (Container) بوصفها نسخة حية وقابلة للتشغيل من هذه الصورة؛ إضافة إلى Docker Compose، وهي أداة تنسيق تتيح إدارة عدة حاويات مترابطة معاً ضمن ملف تصريحي واحد. وقد وُظِّف هذا المبدأ في المشروع لتغليف كل خدمة من خدماته — من نظام الملفات الموزّع، مروراً بقناة البث ومحرّك المعالجة، وصولاً إلى الخدمة الخلفية وقاعدة البيانات — ضمن حاوية مستقلة، بينما استُخدم Docker Compose لتشغيل حزمة الخدمات كوحدة متكاملة على كل عقدة من عقد العنقود، بما أتاح إعادة إنتاج البيئة بالكامل على أي عقدة أخرى دون إعدادات معقّدة.

ويمكن تلخيص مبررات اعتماد Docker في هذا المشروع بأربع فوائد رئيسية: تناسق البيئات، إذ يضمن تشغيل الكود نفسه بالبيئة نفسها عبر جميع مراحل التطوير والاختبار والتشغيل، مما يقلّل من مشكلات التوافق بين الأجهزة المختلفة؛ العزل والمرونة، حيث يعمل كل مكوّن من مكوّنات النظام الموزّع بشكل مستقل عن غيره، بما يبسّط عمليات الصيانة والتوسعة دون التأثير على بقية الخدمات؛ قابلية إعادة الإنتاج، إذ يمكن إعادة إنشاء أي عقدة من عقد النظام بالدقة نفسها بالاعتماد على الصور الموثَّقة مسبقاً؛ إضافة إلى ملاءمته الطبيعية للأنظمة الموزّعة، وهي الخاصية التي استُثمرت مباشرة في توزيع خدمات النظام (كعقد Hadoop وعمّال Spark) عبر عقد العنقود الثلاث.

## React

React.js هي مكتبة مفتوحة المصدر لتطوير واجهات المستخدم باستخدام لغة JavaScript، طوّرتها شركة Meta، وترتكز على مبدأ المكوّنات (Components)، حيث تُقسَّم واجهة المستخدم إلى وحدات مستقلة يعالج كل منها منطق عرضه الخاص، بما يسهّل إعادة استخدامها ودمجها مع بعضها. وتعتمد React أيضاً على ما يُعرف بـ"الشجرة الافتراضية" (Virtual DOM)، وهي نسخة خفيفة من شجرة العرض الفعلية للمتصفح تُستخدم للمقارنة بين الحالة القديمة والجديدة للواجهة قبل تحديثها فعلياً، مما يقلّل من عمليات إعادة الرسم غير الضرورية ويرفع من سرعة استجابة الواجهة.

اعتُمدت React لبناء لوحة التحكم بالكامل، مستفيدة من تدفّق البيانات أحادي الاتجاه الذي تنتهجه المكتبة، والذي يسهّل تتبّع مصدر أي تغيّر في حالة الواجهة، إضافة إلى اعتماد أسلوب المكوّنات الوظيفية (Functional Components) المبني على الخطاطيف (Hooks) كـuseState وَuseEffect، بدلاً من الأسلوب الصنفي (Class-based) الأقدم، بما نتج عنه كود أكثر إيجازاً ووضوحاً. وقد رُوعي في تنظيم بنية الكود الفصل الواضح بين الصفحات (Pages) التي تمثّل الوحدات الوظيفية الكبرى للوحة التحكم، والمكوّنات القابلة لإعادة الاستخدام (Components) التي تخدم أكثر من صفحة في آنٍ واحد، بما يعكس التزاماً بمبدأ فصل الاهتمامات (Separation of Concerns) على مستوى الواجهة الأمامية.

## FastAPI

FastAPI هو إطار عمل حديث ومفتوح المصدر مخصَّص لبناء واجهات برمجة التطبيقات (APIs) بلغة Python، ويتميّز بأدائه العالي مقارنة بالأطر التقليدية الأخرى في اللغة نفسها، نظراً لبنائه على مكتبتي Starlette للتعامل مع طبقة الشبكة، وPydantic للتحقّق من صحة البيانات وتوصيفها، مع اعتماده الكامل على النمط غير المتزامن (Asynchronous) في معالجة الطلبات، بما يتيح خدمة عدد كبير من الاتصالات المتزامنة بكفاءة دون حجب الخادم.

اعتُمد هذا الإطار لبناء الخدمة الخلفية للنظام لسببين رئيسيين: الأول هو دعمه المدمج لبروتوكول WebSocket دون الحاجة لمكتبات إضافية، وهو ما استُثمر مباشرة في بناء قناة البث اللحظي للتنبيهات نحو لوحة التحكم؛ والثاني هو التحقّق التلقائي من صحة البيانات الواردة عبر نماذج Pydantic (كنموذجي التنبيه Alert وحكم المحلّل VerdictUpdate)، بما يقلّل من الأخطاء المحتملة الناتجة عن بيانات مشوَّهة قبل وصولها إلى منطق معالجة الطلب، ويجعل توثيق واجهة البرمجة ذاتياً ومتوافقاً مع مواصفة OpenAPI دون جهد إضافي.

الفصل الثامن

# الاختبارات

*نقوم في هذا الفصل بعرض نتائج الاختبارات و تلبيتها للمتطلبات.*

## 1.8- الاختبارات الوظيفية

### 1.1.8- اختبارات الوحدة

تم تنفيذ مجموعة من الاختبارات البرمجية التلقائية للخدمة الخلفية (Back-end) لضمان جودة واستقرار منطق الأعمال المسؤول عن استقبال التنبيهات الأمنية وإدارتها. اعتمدت هذه الاختبارات على إطار العمل pytest بالتكامل مع مكتبة httpx واستخدام أداة TestClient المدمجة في FastAPI، وهدفت إلى التأكد من صحة عمل كل وحدة من وحدات النظام بشكل منفصل ومعزول عن قاعدة البيانات الفعلية، تفادياً لأي تأثير جانبي على البيانات الحقيقية المخزّنة في PostgreSQL أثناء التطوير أو الاختبار.

تم تغطية الأنواع التالية من الاختبارات:

- اختبارات الصحة والمصادقة (Health & Authentication Tests): للتحقق من صحة استجابة نقاط النهاية الأساسية، ونجاح تسجيل الدخول ببيانات صحيحة ورفضه ببيانات خاطئة، بالإضافة إلى التأكد من حماية النقاط الحساسة (كحظر عناوين IP) برمز مصادقة (Token) صالح، ورفض أي طلب لا يحمل هذا الرمز.

- اختبارات منطق التنبيهات (Alerts Logic Tests): لفحص السلوك الجوهري لآلية استقبال التنبيهات القادمة من محرك Spark، وتحديداً اختبار منطق الدمج (Upsert Logic) الذي يجمّع التنبيهات المتكررة لنفس عنوان IP ونوع الهجوم خلال نافذة زمنية مدتها 60 ثانية بدلاً من تكرارها كسجلات منفصلة، وهو ما يمنع إغراق لوحة التحكم بتنبيهات مكررة لهجوم واحد مستمر.

- اختبارات صحة المدخلات (Validation Tests): للتأكد من رفض النظام لأي طلب يفتقد حقلاً إلزامياً (مثل عنوان المصدر أو مستوى الخطورة) بإرجاع رمز الحالة 422، وكذلك رفض أي قيمة غير معرّفة لحالة المراجعة (Verdict) بإرجاع رمز الحالة 400.

▪ اختبارات الدوال المساعدة (Helper Function Tests): للتحقق من صحة دالة تصنيف عناوين IP إلى داخلية أو خارجية، وضمان أن أي مدخل غير صالح (Malformed Input) يُعامَل بشكل آمن كعنوان داخلي بدلاً من التسبب بانهيار النظام أو استدعاء غير ضروري لخدمة السمعة الخارجية (AbuseIPDB).

اعتمدت عملية اختبار النظام على أداة unittest.mock.patch لمحاكاة الاتصال بقاعدة بيانات PostgreSQL عبر استبدال دالة get_db() بكائن وهمي (Mock Object)، مما وفّر تغطية كاملة لمنطق الأعمال دون الحاجة لتشغيل قاعدة بيانات فعلية أثناء الاختبار، وهو ما يماثل دور أداة DataJpaTest في الأنظمة المبنية على Spring Boot. كما تم استخدام fixtures خاصة بـ pytest لحقن رمز مصادقة صالح مباشرة في الذاكرة، لاختبار النقاط المحمية دون المرور بدورة تسجيل الدخول الكاملة في كل اختبار.

بلغ إجمالي عدد الاختبارات المنفّذة 19 اختباراً، جميعها نجحت دون أي فشل .

## 2.8 اختبارات الأداء – اختبار الحمل

مقدمة عامة:

تُعد اختبارات الأداء من المراحل الجوهرية في تقييم كفاءة نظام Smart-IDPS، لا سيما وأنه نظام أمني لحظي (Real-time) يعتمد على استقبال ومعالجة تنبيهات أمنية متدفقة من محرك التحليل (Spark Streaming) وتسجيلها عبر الخدمة الخلفية (FastAPI). لا تكفي اختبارات الوظائف وحدها لضمان جاهزية النظام؛ إذ لا بد من اختبار مدى تحمّله للأحمال المتزايدة من الطلبات المتزامنة، واستقراره عند الوصول إلى حدود الاستخدام القصوى. تهدف هذه الاختبارات إلى قياس مجموعة من المؤشرات الأساسية:

▪ زمن الاستجابة (Response Time): كم من الوقت يحتاج النظام لمعالجة الطلب الواحد.

▪ معدل الإنتاجية (Throughput): عدد الطلبات التي يستطيع النظام معالجتها فعلياً في الثانية الواحدة.

▪ استهلاك الموارد (CPU/Memory Usage): كيف يتغيّر استهلاك النظام للموارد تحت الضغط.

- نسبة الأخطاء (Error Rate): ما نسبة الطلبات التي تفشل أثناء الحمل المرتفع.

الأداة المستخدمة

تم استخدام أداة Apache JMeter (الإصدار 5.6.3) لتنفيذ جميع اختبارات الحمل، نظراً لدعمها الكامل لمحاكاة سيناريوهات استخدام واقعية بأسلوبين مختلفين ومتكاملين: تثبيت معدل ثابت للطلبات في الثانية (Constant Throughput Timer)، أو محاكاة عدد من الطلبات المتزامنة (Concurrent Requests) الموجَّهة نحو نقطة النهاية، بحيث يمثّل كل طلب مصدر إرسال مستقل يعمل بالتوازي مع البقية (Thread Groups). كما تدعم الأداة قراءة بيانات مختلفة لكل طلب من ملفات خارجية (CSV Data Set Config)، وتولّد تلقائياً تقارير HTML تفاعلية تفصيلية بعد كل اختبار. تم تنفيذ جميع الاختبارات بصيغة التشغيل غير الرسومية (Non-GUI Mode) لضمان دقة القياس دون استهلاك موارد إضافية من واجهة المستخدم الرسومية.

تحديد نقطة النهاية المستهدفة

من بين نقاط النهاية المتعددة في الخدمة الخلفية، تم اختيار POST /api/alerts كهدف رئيسي لاختبار الحمل. يعود هذا الاختيار إلى طبيعة النظام كنظام أمني: هذه النقطة هي التي يعتمد عليها محرك التحليل (Spark) لتسجيل كل تنبيه فور اكتشافه، وبالتالي فإن أي فشل أو تأخر في معالجتها يحمل أثراً أمنياً مباشراً (احتمال تأخر أو فقدان تسجيل هجوم حقيقي)، بخلاف نقاط القراءة (GET) التي يقتصر استخدامها على عرض البيانات في لوحة التحكم .

الاختبار الأول : اختبار الحمل بمعدل ثابت

منهجية الاختبار :

يعتمد هذا الأسلوب على تثبيت معدل مُحدَّد مسبقاً من الطلبات في كل ثانية، بحيث تتولى أداة الاختبار (JMeter) عبر مكوّن Constant Throughput Timer ضبط سرعة الإرسال بدقة لتطابق الرقم المطلوب، بغض النظر عن سرعة استجابة النظام. بمعنى آخر: الجهة المختبِرة هي من تتحكم بالسرعة المرسَلة، وليس النظام المختبَر.

الغاية من هذا الأسلوب هي الإجابة عن سؤال محدد: "هل النظام قادر على استيعاب معدل مُحدَّد بعينه من الطلبات، والحفاظ على هذا المعدل دون تراكم أو تأخر، طوال فترة زمنية ممتدة؟"

عند إرسال دفعة من الطلبات في ثانية معينة، هناك احتمالان: إما أن ينجح النظام في معالجة هذه الدفعة بالكامل قبل وصول الدفعة التالية (فيستمر المعدل الفعلي مطابقاً للمطلوب)، أو أن يعجز عن ذلك بسبب بطء المعالجة، فتتراكم الطلبات غير المنجَزة فوق الطلبات الجديدة الواردة (Queueing)، مما يؤدي إلى انخفاض المعدل الفعلي المنجَز عن المعدل المطلوب، وارتفاع تدريجي في زمن الاستجابة.

اشتقاق الحمل المتوقع من بيانات حقيقية

بدلاً من الاعتماد على تقدير نظري، تم استخراج معدل التنبيهات الفعلي من سجلات قاعدة البيانات نفسها، عبر تجميع عدد التنبيهات المسجَّلة لكل دقيقة خلال فترات التشغيل السابقة. أظهر الاستعلام أن النظام سجّل خلال إحدى فترات الهجوم الفعلية ذروة بلغت 1,434 تنبيهاً خلال دقيقة واحدة، أي ما يعادل تقريباً 24 طلباً في الثانية. اعتُمد هذا الرقم كأساس لتصميم مراحل اختبار الحمل التالية.

خطة الاختبار

تم تصميم سيناريو اختبار تصاعدي على ثلاث مراحل، كل مرحلة بمعدل ثابت لمدة 3 دقائق متواصلة:

| المرحلة | المعدل المطلوب | التبرير |
|---|---|---|
| الأولى | 25طلب /ثانية | مطابقة تقريباً للذروة الحقيقية المسجَّلة(24/s) |
| الثانية | 100 طلب /ثانية | أربعة أضعاف الذروة الحقيقية، لاختبار هامش الأمان |
| الثالثة | 200طلب/ ثانية | ثمانية أضعاف الذروة الحقيقية، لتحديد نقطة التشبع |

التجربة الأولى — 25 طلب/ثانية (مطابقة للذروة الحقيقية المسجَّلة):

أظهرت النتائج أداءً ممتازاً؛ تم إنجاز 4,514 طلباً بمعدل فعلي 25.1 طلب/ثانية (مطابق تماماً للمطلوب)، بمتوسط زمن استجابة لم يتجاوز 21 ميلي ثانية، وحد أقصى 154 ميلي ثانية، دون تسجيل أي خطأ. كما أظهرت مراقبة الموارد أن استهلاك المعالج لم يتجاوز 0.25%.

APDEX (Application Performance Index)

| Apdex | T (Toleration threshold) | F (Frustration threshold) | Label |
|---|---|---|---|
| 1.000 | 500 ms | 1 sec 500 ms | Total |
| 1.000 | 500 ms | 1 sec 500 ms | POST /api/alerts |

Requests Summary



Statistics

| Requests | Executions | | | Response Times (ms) | | | | | | | Throughput | Network (KB/sec) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Label | #Samples | FAIL | Error % | Average | Min | Max | Median | 90th pct | 95th pct | 99th pct | Transactions/s | Received | Sent |
| Total | 4514 | 0 | 0.00% | 21.47 | 12 | 154 | 18.00 | 32.00 | 35.00 | 46.85 | 25.12 | 6.34 | 6.12 |
| POST /api/alerts | 4514 | 0 | 0.00% | 21.47 | 12 | 154 | 18.00 | 32.00 | 35.00 | 46.85 | 25.12 | 6.34 | 6.12 |

التجربة الثانية — 100 طلب/ثانية:

بدأت أولى مؤشرات الضغط بالظهور؛ تراجع معدل الطلبات المنجزة فعلياً إلى 55.3 طلب/ثانية (أقل من نصف المطلوب)، وارتفع متوسط زمن الاستجابة إلى 1,721 ميلي ثانية، بينما بقي استهلاك المعالج منخفضاً نسبياً (13.09%) ونسبة الأخطاء صفراً.

التجربة الثالثة — 200 طلب/ثانية:

تراجع معدل الطلبات المنجزة فعلياً أكثر (47.31 طلب/ثانية)، وارتفع متوسط زمن الاستجابة إلى 4,102.72 ميلي ثانية، مع وصول الشريحة المئوية 99 إلى 61,630 ميلي ثانية، وبقاء نسبة الأخطاء عند صفر.

APDEX (Application Performance Index)

| Apdex | T (Toleration threshold) | F (Frustration threshold) | Label |
|---|---|---|---|
| 0.211 | 500 ms | 1 sec 500 ms | Total |
| 0.211 | 500 ms | 1 sec 500 ms | POST /api/alerts |

Requests Summary



Statistics

| Requests | Executions | | | Response Times (ms) | | | | | | | | Throughput | Network (KB/sec) | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Label | #Samples | FAIL | Error % | Average | Min | Max | Median | 90th pct | 95th pct | 99th pct | | Transactions/s | Received | Sent |
| Total | 8658 | 0 | 0.00% | 4102.72 | 68 | 62558 | 1785.00 | 2701.00 | 8691.35 | 61630.41 | | 47.31 | 12.34 | 11.55 |
| POST /api/alerts | 8658 | 0 | 0.00% | 4102.72 | 68 | 62558 | 1785.00 | 2701.00 | 8691.35 | 61630.41 | | 47.31 | 12.34 | 11.55 |

ارتفاع حاد في زمن الاستجابة دون ارتفاع مقابل في استهلاك المعالج — بالرجوع إلى إعداد مجموعة اتصالات قاعدة البيانات (Connection Pool) المطبَّق على مستوى الخدمة الخلفية، والمحدَّد بحجم أقصى قدره 50 اتصالاً متزامناً (maxconn=50) في psycopg2.pool.ThreadedConnectionPool)، مع آلية إعادة محاولة دورية (while True: time.sleep(0.05)) للطلبات التي تتجاوز هذا الحد، بدلاً من رفضها.

هذا الإعداد يفرض سقفاً فعلياً وثابتاً لعدد الطلبات التي يمكن معالجتها فعلياً في آنٍ واحد، بصرف النظر عن معدل الطلبات الواردة من الخارج. فعندما يكون المعدل المطلوب ضمن هذا السقف أو قريباً منه (كما في حالة 25 طلب/ثانية)، تُعالَج جميع الطلبات دون انتظار يُذكر، ويتطابق المعدل الفعلي مع المطلوب. أما عند تجاوز المعدل المطلوب لهذا السقف بشكل ملحوظ (100 و200 طلب/ثانية)، فإن الطلبات الزائدة عن الاتصالات الخمسين المتاحة تدخل في طابور انتظار فعلي، وتظل "معلَّقة" (لا تُعالَج ولا تُرفَض) إلى حين تحرُّر أحد الاتصالات المستخدَمة. وبما أن كل اتصال يُستخدَم لعملية إدراج (INSERT) واحدة قبل تحرره، فإن معدل تحرُّر الاتصالات — وليس معدل وصول الطلبات — هو ما يحدد فعلياً السرعة القصوى التي يمكن للنظام أن يعالج بها الطلبات، بصرف النظر عن مدى ارتفاع الضغط الخارجي.

هذا يفسّر أيضاً سبب انخفاض المعدل الفعلي عند 200 طلب/ثانية (47.31) مقارنة بحالة 100 طلب/ثانية (55.3)، رغم ارتفاع الضغط المطلوب: فكلما زاد عدد الطلبات المتنافسة على الاتصالات الخمسين المحدودة، طال زمن انتظار كل طلب في الطابور، وبالتالي انخفض العدد الإجمالي للطلبات القادرة على إتمام دورتها الكاملة (إرسال، انتظار اتصال، معالجة، تحرير الاتصال) خلال المدة الثابتة للاختبار (3 دقائق).

بناءً على هذا التحليل، فإن هذا النمط ليس ظاهرة عابرة مرتبطة بظروف تشغيل معينة، بل هو سلوك بنيوي متوقع الحدوث في كل مرة يتجاوز فيها عدد الطلبات المتزامنة الفعلية حجم مجموعة الاتصالات المعرَّفة (50 اتصالاً)، بصرف النظر عن توقيت الاختبار أو الحمل الكلي على النظام. ويؤكد ذلك تطابق النمط نفسه في الاختبار المستقل اللاحق القائم على أسلوب الطلبات المتزامنة، حيث استقر معدل الإنتاجية عند قيمة ثابتة (~92–95 طلب/ثانية) بصرف النظر عن عدد الطلبات المتزامنة المرسَلة (50، 200، أو 500)،

الاختبار الثاني : اختبار الحمل بأسلوب الطلبات المتزامنة

منهجية الاختبار: أسلوب الطلبات المتزامنة (Concurrent Requests)

اعتُمد في هذا الاختبار أسلوب محاكاة عدد محدد من الطلبات المتزامنة (Concurrent Requests) الموجَّهة نحو نقطة النهاية، حيث تمثّل كل مجموعة طلبات (Thread) مصدر إرسال مستقل يعمل بالتوازي مع بقية المصادر طوال مدة الاختبار. يحاكي هذا الأسلوب سيناريو أقرب للواقع الأمني الفعلي: فحين يقع هجوم حقيقي (كهجوم DDoS)، لا تصل التنبيهات إلى النظام بمعدل منتظم وثابت من مصدر واحد، بل على شكل موجة ضغط تزامنية ناتجة عن معالجة محرك التحليل (Spark) لكميات كبيرة من حركة الشبكة في اللحظة نفسها. لذلك فإن اختبار قدرة النظام على استيعاب عدد كبير من الطلبات المتزامنة أدق تعبيراً عن الحالة الحرجة التي صُمم النظام أصلاً لمواجهتها.

تجهيز بيانات الاختبار

لتفادي إرسال طلبات متطابقة لا تعكس سلوكاً واقعياً، تم توليد ملف بيانات (CSV Data Set Config) يحتوي على 800 سجل تنبيه متنوع، بحيث يقرأ كل طلب سجلاً مختلفاً من الملف تلقائياً أثناء التنفيذ. صُمم هذا الملف ليحاكي توزيعاً واقعياً لحركة الهجمات:

- 20% من السجلات تمثّل عناوين IP متكررة من قائمة مهاجمين معروفين (محاكاة لمهاجم واحد يكرر النشاط عدة مرات).
- 80% من السجلات عبارة عن عناوين IP عشوائية موزّعة على ثلاثة أنواع من الهجمات (DDoS، Brute-Force، DoS)، كل منها بمستوى خطورة مناسب لطبيعته.

هذا التنوع يضمن أن الاختبار لا يقيس فقط سرعة الاستجابة، بل يُبقي منطق الدمج (Upsert Logic) الخاص بالنظام — والذي يجمّع التنبيهات المتكررة لنفس عنوان IP خلال 60 ثانية — يعمل في سياق واقعي أثناء الاختبار، بدلاً من تعطيله بالكامل عبر توليد عناوين فريدة لكل طلب.

خطة الاختبار

تم تنفيذ الاختبار على ثلاث مراحل تصاعدية من حيث عدد الطلبات المتزامنة المرسَلة إلى نقطة النهاية: 50، ثم 200، ثم 500 طلباً متزامناً، حيث تُكرَّر عملية الإرسال عدة مرات متتالية طوال مدة كل اختبار، بهدف رفع الضغط التزامني تدريجياً وتحديد النقطة التي يبدأ عندها أداء النظام بالتدهور.

التجربة الأولى — 50 طلباً متزامناً:

سجّل النظام أداءً مستقراً تماماً عند هذا المستوى من الحمل؛ بلغ معدل الإنتاجية 92.5 (Throughput) طلباً في الثانية، بمتوسط زمن استجابة لم يتجاوز 51 ميلي ثانية، وحد أقصى بلغ 98 ميلي ثانية فقط، دون تسجيل أي طلب فاشل (0.00% نسبة أخطاء). يعكس هذا الأداء قدرة النظام على التعامل بسلاسة تامة مع حمل يقارب ضعفي الحمل الحقيقي الأقصى المرصود سابقاً في سجلات النظام.

```
~/apache-jmeter-5.6.3/bin/jmeter -n -t ids_load_test.jmx -l results.jtl -e -o
report_html
WARN StatusConsoleListener The use of package scanning to locate plugins is
deprecated and will be removed in a future release
WARN StatusConsoleListener The use of package scanning to locate plugins is
deprecated and will be removed in a future release
WARN StatusConsoleListener The use of package scanning to locate plugins is
deprecated and will be removed in a future release
WARN StatusConsoleListener The use of package scanning to locate plugins is
deprecated and will be removed in a future release
Creating summariser <summary>
Created the tree successfully using ids_load_test.jmx
Starting standalone test @ August 5, 2026 10:45:39 PM UTC (1785969939848)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message
on port 4445
summary =  1000 in 00:00:11 =  92.5/s Avg:  51 Min:  10 Max:  98 Err:  0 (0.00%)
Tidying up ...   @ August 5, 2026 10:45:51 PM UTC (1785969951093)
... end of run
```

التجربة الثانية — 200 طلباً متزامناً:

عند رفع عدد المستخدمين المتزامنين إلى 200، بدأت أولى مؤشرات التدهور بالظهور بشكل واضح. اللافت أن معدل الإنتاجية بقي شبه ثابت (95.0 طلب/ثانية) رغم مضاعفة الحمل أربعة أضعاف، إلا أن متوسط زمن الاستجابة ارتفع بشكل حاد إلى 645 ميلي ثانية (أي بمعدل يزيد عن 12 ضعفاً مقارنة بالتجربة الأولى)، ووصل الحد الأقصى لزمن الاستجابة إلى 9,811 ميلي ثانية (ما يقارب 10 ثوانٍ) لبعض الطلبات، مع بقاء نسبة الأخطاء عند صفر.

```
~/apache-jmeter-5.6.3/bin/jmeter -n -t ids_load_test_200.jmx -l results_200.jtl -e -o
report_html_200
WARN StatusConsoleListener The use of package scanning to locate plugins is
deprecated and will be removed in a future release
Creating summariser <summary>
Created the tree successfully using ids_load_test_200.jmx
Starting standalone test @ August 6, 2026 9:05:21 AM UTC (1786007121548)
Waiting for possible Shutdown/StopTestNow/HeapDump/ThreadDump message
on port 4445
summary +   671 in 00:00:08 =  84.0/s Avg:  353 Min:   49 Max:  3537 Err:    0
(0.00%) Active: 92 Started: 150 Finished: 58
summary +  1329 in 00:00:13 = 101.7/s Avg:  792 Min:   11 Max:  9811 Err:    0 (0.00%)
Active: 0 Started: 200 Finished: 200
summary =  2000 in 00:00:21 =  95.0/s Avg:  645 Min:   11 Max:  9811 Err:    0 (0.00%)
Tidying up ...   @ August 6, 2026 9:05:43 AM UTC (1786007143169)
... end of run
```

التجربة الثالثة — 500 مستخدماً متزامناً:

عند الوصول إلى 500 طلب متزامن — أي عشرة أضعاف التجربة الأولى — تأكد النمط الملحوظ بشكل قاطع. استقر معدل الإنتاجية عند 92.0 طلب/ثانية، أي بلا أي تحسّن فعلي مقارنة بالتجربتين السابقتين رغم الزيادة الهائلة في الحمل المرسَل. في المقابل، تضاعف متوسط زمن الاستجابة إلى 1,753 ميلي ثانية، ووصل الحد الأقصى إلى 16,173 ميلي ثانية (أكثر من 16 ثانية) لبعض الطلبات. نسبة الأخطاء عند صفر بالكامل — أي أن النظام لم يفقد أو يرفض ولا طلباً واحداً من أصل جميع الطلبات المرسَلة، رغم الضغط الشديد.

```
~/apache-jmeter-5.6.3/bin/jmeter -n -t ids_load_test_500.jmx
-l results_500.jtl -e -o report_html_500
WARN StatusConsoleListener The use of package scanning
to locate plugins is deprecated and will be removed in a
future release
Creating summariser <summary>
Created the tree successfully using ids_load_test_200.jmx
Starting standalone test @ August 6, 2026 9:05:21 AM UTC
(1786007121548)
Waiting for possible
Shutdown/StopTestNow/HeapDump/ThreadDump message
on port 4445
summary =  4000 in 00:00:21 =   95.0/s Avg:  1,753 Min:   11
Max:  16.173 Err:    0 (0.00%)
Tidying up ...   @ August 6, 2026 9:05:43 AM UTC
(1786007143169)
... end of run
```

جدول ملخص المرحلة الثانية:

| عدد الطلبات المتزامنة | معدل الإنتاجية Throughput | متوسط الاستجابة | الحد الأقصى | نسبة الأخطاء |
|---|---|---|---|---|
| 50 | 92.5طلب/ثانية | 51ms | 98ms | 0.00% |
| 200 | 92.0طلب/ثانية | 645ms | 9,811ms | 0.00% |
| 500 | 92.0طلب/ثانية | 1,753ms | 16,173ms | 0.00% |

تحليل النتائج

نلاحظ ثبات معدل الإنتاجية عند حوالي 92–95 طلباً في الثانية بصرف النظر عن عدد الطلبات المتزامنة (سواء كانوا 50 أو 200 أو 500). يمثّل هذا النمط التوقيع الكلاسيكي لظاهرة تشبع الموارد (Resource Saturation): النظام وصل إلى الحد الأقصى لقدرته على معالجة الطلبات المتزامنة، وأي زيادة إضافية في عدد المستخدمين لا تزيد من الإنتاجية الفعلية، بل تزيد فقط من زمن انتظار كل طلب في طابور المعالجة.

لتحديد السبب الجذري، تمت مراقبة استهلاك المعالج (CPU) لحاوية الخدمة الخلفية بشكل متزامن مع تنفيذ الاختبار عبر سكريبت مراقبة مخصص. أظهرت النتائج أن استهلاك المعالج بقي منخفضاً جداً (أقل من 1%) طوال فترة الاختبار، حتى في اللحظات التي تجاوز فيها زمن الاستجابة 9 ثوانٍ.

هذا التباين الحاد بين ارتفاع زمن الاستجابة وانخفاض استهلاك المعالج بشكل متزامن يستبعد أن يكون السبب متعلقاً بنقص في موارد المعالجة، ويؤكد أن السبب الفعلي هو استنفاد مجموعة اتصالات قاعدة البيانات (Connection Pool Exhaustion). فبعد تطبيق آلية تجميع الاتصالات (psycopg2.pool.ThreadedConnectionPool) بحجم أقصى قدره 50 اتصالاً — كحل لمشكلة استهلاك المعالج المرتفع التي ظهرت في مرحلة اختبار سابقة على نقطة النهاية GET — أصبحت الطلبات الزائدة عن هذه السعة تدخل في حالة انتظار فعلي (عبر آلية إعادة المحاولة الدورية) إلى حين توفّر اتصال متاح، بدلاً من أن تُرفَض أو تستهلك موارد معالجة إضافية. هذا يفسّر ارتفاع زمن الاستجابة الحاد دون أي ارتفاع مقابل في استهلاك المعالج، إذ إن الطلب المعلَّق لا "يعمل" فعلياً بل "ينتظر" دوره.

الحلول المحتملة وحدودها

بناءً على هذا التشخيص، تم بحث حلين محتملين لرفع سعة النظام:

الحل الأول — رفع الحد الأقصى لحجم مجموعة الاتصالات (maxconn):

يبدو هذا الحل مباشراً، إلا أنه محكوم بقيد أعمق: خادم قاعدة البيانات PostgreSQL نفسه يفرض حداً أقصى لعدد الاتصالات المسموح بها في آنٍ واحد (عبر إعداد max_connections، والذي يكون عادة 100 اتصال بالإعداد الافتراضي).

```
ubuntu@k3:~$
ubuntu@k3:~$ cd ids-project/
ubuntu@k3:~/ids-project$ sudo docker exec postgres psql -U ids_user -d ids_db -c "SHOW max_connections;"

 max_connections
-----------------
 100
(1 row)
```

وبالتالي، فإن رفع قيمة maxconn في التطبيق دون التأكد من توافقها مع هذا القيد على مستوى قاعدة البيانات لن يحل المشكلة الجذرية، بل قد ينقلها إلى مستوى آخر (رفض الاتصالات من قبل قاعدة البيانات نفسها).

الحل الثاني — استخدام مكتبة اتصال غير متزامنة (Asynchronous) مثل asyncpg بدلاً من psycopg2:

تتميز المكتبات غير المتزامنة بأنها لا تحجز خيط معالجة (Thread) كاملاً طوال فترة انتظار الاستجابة من قاعدة البيانات، بل تتيح استخدام نفس الموارد لمعالجة طلبات أخرى أثناء الانتظار، مما يحسّن كفاءة استغلال الموارد المتاحة. إلا أن هذا الحل لا يلغي القيد الأساسي المفروض من قاعدة البيانات نفسها (max_connections)؛ فسواء كان الاتصال متزامناً أو غير متزامن، يبقى العدد الأقصى للاتصالات الفعلية المسموح بفتحها مع قاعدة البيانات محكوماً بنفس الحد. كما يتطلب تطبيق هذا الحل إعادة هيكلة جوهرية للكود المصدري بالكامل (تحويل جميع الدوال والاستدعاءات إلى النمط غير المتزامن)، ما يزيد من تعقيد الصيانة واحتمالية الأخطاء.

# الخاتمة

الملحق ب

# المراجع وتنسيق قائمتها

# المراجع