ChatGPT

# الفرق بين تطوير أنظمة SaaS والأنظمة التقليدية (دراسة مقارنة شاملة)

في عالم البرمجيات الحديثة، يبرز نموذج **البرمجيات كخدمة (SaaS)** كنهج مختلف جذريًا عن الأنظمة البرمجية التقليدية (التي عادة تكون حلولًا مخصصة تُنشر محليًا لدى العميل أو في مراكز بيانات خاصة به). يقدم SaaS التطبيقات عبر الإنترنت كخدمة مستضافة لدى مزود خارجي، مما يلغي حاجة العملاء لإدارة البنية التحتية أو تثبيت البرمجيات محليًا [1] . في المقابل، تعتمد الأنظمة التقليدية (On-Premise) على تثبيت البرمجيات وتشغيلها ضمن بيئة العميل الخاصة، مع تحمل العميل مسؤولية الصيانة والتحديث والبنية التحتية. سنستعرض في هذا التقرير **فروق التطوير بين SaaS والأنظمة التقليدية عبر جميع المراحل بدءًا من تحليل المتطلبات وحتى التنفيذ**، مع التركيز على مثال **نظام محاسبي** أو **منصة متجر إلكتروني متعدد التجار** كنموذج توضيحي. سنغطي منهجية جمع المتطلبات، التصميم المعماري والبنية التحتية، إدارة المستخدمين والبيانات وتعدد المستأجرين، أساليب النشر والتحديث والصيانة، أمثلة واقعية، الأخطاء الشائعة في تصميم SaaS، التحديات الأمنية والامتثال، وتأثير هذه الفروق على التكلفة وقابلية التوسع وتجربة المستخدم.

## 1. منهجية التحليل وجمع المتطلبات

**اختلاف نطاق المتطلبات والتخصيص:** في المشاريع التقليدية (الأنظمة المخصصة أو المحلية)، غالبًا ما يتم جمع المتطلبات من عميل محدد أو مؤسسة واحدة بهدف **تفصيل الحل لاحتياجاتها الخاصة** . تكون عملية التحليل معمقة في فهم إجراءات عمل ذلك العميل وقوانينه الداخلية، مما يؤدي عادةً إلى نظام شديد التخصيص لتلبية متطلبات محددة جدًا [2] . بالمقابل، يتطلب تطوير منتج SaaS اعتماد **منهجية منتجات أشمل** ؛ حيث يجب تصميم النظام ليلبي احتياجات شريحة واسعة من العملاء (الشركات أو المستخدمين) عبر قطاعات مختلفة. لذلك يُنظر إلى **متطلبات تطبيقات SaaS بمنظور الميزات والقابليات العامة وسهولة الاستخدام** بحيث تغطي معظم السيناريوهات الشائعة [2] ، ويتم تقليل التخصيص الخاص بكل عميل لصالح نموذج موحد قابل للاعداد والتكوين. على سبيل المثال، عند بناء **منصة محاسبية SaaS** ، يتم جمع المتطلبات العامة للمحاسبة (الفوترة، إدارة الحسابات، الضرائب المتنوعة) بطريقة **تتيح تكوين النظام لكل شركة** بدلًا من بناء منطق مخصص لكل شركة كما قد يحدث في نظام محاسبي داخلي موجه لشركة واحدة.

**النهج التقليدي مقابل المرن:** مشاريع الأنظمة التقليدية كثيرًا ما اتبعت منهجيات شبيهة بالشلال (Waterfall) حيث يتم تعريف المتطلبات بالكامل منذ البداية ثم التصميم فالتنفيذ دفعة واحدة. في المقابل، تميل مشاريع SaaS الحديثة إلى اتباع **منهجيات مرنة (Agile)** نظرًا لكون SaaS عبارة عن **منتج مستمر التطور**. يتطلب SaaS **إصدار تحسينات متكررة** واستقبال التغذية الراجعة من قاعدة العملاء المتنوعة، ومن ثم تعديل المتطلبات دوريًا وإعادة ترتيب أولويات الميزات. هذا يعني أن **عملية جمع المتطلبات في SaaS مستمرة** طوال دورة حياة المنتج (من خلال استطلاع آراء العملاء، تحليل السوق، تتبع استخدام الميزات) لضمان بقاء الخدمة جذابة لشريحة واسعة من المستأجرين.

**إشراك أصحاب المصلحة ومتطلبات** multi-tenancy ***: من الفروق الأساسية أيضًا أنه في *SaaS يجب منذ مرحلة التحليل *أخذ تعددية المستأجرين بعين الاعتبار** . أي يجب طرح أسئلة مثل: كيف سيُستخدم النظام من قبل عدة عملاء *(شركات مختلفة)* بشكل متزامن؟ ما هي حدود التخصيص المسموحة لكل عميل؟ وكيف سنعزل بيانات كل عميل؟ . يؤكد الخبراء على أهمية التفكير في **تعددية المستأجرين منذ اليوم الأول** وعدم تأجيلها، لأن عدم تضمين مفهوم "المستأجر" في تصميم المتطلبات من البداية قد يجعل إضافة التعددية لاحقًا أمرًا بالغ الصعوبة [3] . في مشروع **متجر إلكتروني متعدد التجار** مثلًا، ينبغي أثناء التحليل تحديد المتطلبات العامة لكل متجر (كتعريف منتجاته وأسعاره وشحنه) ضمن إطار منصة مشتركة، بدل بناء متجر منفصل لكل تاجر. كذلك **يجب التنسيق المبكر بين فرق العمل المختلفة** (تطوير، عمليات، دعم) حتى في SaaS لضمان أن المتطلبات غير الوظيفية

(كالأمان والأداء والتوافقية) ملباة لجميع العملاء. باختصار، يركز تحليل *SaaS* على القواسم المشتركة والقابلة للتهيئة، بينما يركز تحليل الأنظمة التقليدية على التفاصيل الخاصة وحلول التفصيل لكل عميل.

## 2. التصميم المعماري والبنية التحتية

**الحوسبة السحابية مقابل البنية المحلية:** يعتمد SaaS عادةً على بنية تحتية سحابية مرنة تستضيفها جهة مزودة للخدمة، مما يعني أن التطبيق مصمم للعمل في **بيئة متعددة المستأجرين عبر الإنترنت**. يكون التصميم المعماري في الغالب **معماريّة متعددة الطبقات** (طبقة واجهة أمامية، طبقة خدمات/واجهات برمجية، طبقة بيانات) قابلة للتوسع أفقيًا لتخدم عددًا كبيرًا من المستخدمين. **تُستخدم الخوادم الافتراضية والحاويات وأنظمة التوازن** لتوزيع الحمل ديناميكيًا بحسب حاجة العملاء. في كثير من الأحيان، يتم **فصل الواجهة الأمامية عن الخلفية** باستخدام تقنيات حديثة؛ فمثلًا تعتمد العديد من تطبيقات SaaS على واجهات ويب تفاعلية مبنية بإطارات عمل JavaScript حديثة (مثل React) متصلة بواجهات API خلفية مبنية على منصات سريعة مثل Node.js لتوفير أداء عالي واستجابة فورية للمستخدمين. في المقابل، تميل الأنظمة التقليدية إلى تصميم معماري أبسط لكل نشر مستقل؛ فقد تكون **تطبيقًا وحيدًا (Monolith)** مثبتًا على خادم مخصص لدى العميل، أو **تطبيق عميل/خادم** ضمن شبكة داخلية. البنية التحتية هنا تكون تحت سيطرة العميل بالكامل – من الخوادم إلى التخزين والشبكات – وتُضبط وفق قدرات وميزانية كل عميل على حدة.

**تعدد المستأجرين (Multi-tenancy) مقابل العزل (Single-tenancy):** إحدى أبرز سمات عمارة SaaS هي دعم **تعدد المستأجرين**، أي أن **عدة عملاء يشتركون في نفس التطبيق والبنية مع عزل منطقي للبيانات لكل عميل** [4]. هناك نماذج معمارية مختلفة لتحقيق ذلك كما هو موضح في الشكل التالي، حيث يمكن تصميم التطبيق وفق نموذج **"المسبح" (Pool)** الذي تتشارك فيه جميع الجهات المستأجرة نفس موارد التطبيق وقاعدة البيانات، أو نموذج **"العزل" (Silo)** حيث يحصل كل عميل على بنية منفصلة بالكامل (قاعدة بيانات أو حتى نسخة تطبيق خاصة)، أو نموذج **وسطي (Bridge)** يجمع بين بعض الموارد المشتركة وبعض العناصر المخصصة لكل مستأجر [5]. اختيار النموذج يؤثر على التكلفة والتعقيد ومستوى العزل الأمني؛ فمثلًا النموذج المشترك أقل كلفة وأبسط صيانة لكنه الأقل عزلة أمنيًا، بينما النموذج المعزول يحقق عزلة تامة لكل عميل لكنه الأعلى كلفة والأكثر تعقيدًا في الإدارة [5].

رسم يوضح ثلاث استراتيجيات معمارية لتعدد المستأجرين في *SaaS*: نموذج الموارد المشتركة بالكامل *(Pool)*، والنموذج الوسيط *(Bridge)* حيث يُفصل بعض ما يتعلق بالعملاء، والنموذج المعزول *(Silo)* بموارد منفصلة لكل عميل. يزداد مستوى العزل الأمني والتعقيد والكلفة عند الانتقال من اليسار *(*المشترك*)* إلى اليمين *(*المعزول*)* [5].

في تصميم SaaS، يُبنى التطبيق عادةً على **قاعدة أكواد واحدة مشتركة لجميع العملاء** لتبسيط الصيانة والتطوير، ويتم تحقيق العزل عن طريق هيكلة البيانات بشكل مناسب (مثل إضافة معرف المستأجر لكل سجل في قاعدة بيانات مشتركة، أو استخدام مخططات Schemas منفصلة لكل مستأجر، أو حتى **قواعد بيانات منفصلة لكل مستأجر في الحالات الحرجة أمنيًا** [6]). أما الأنظمة التقليدية فهي بطبيعتها **أحادية المستأجر**؛ أي أن كل نشر للبرنامج يخدم عميلًا واحدًا، مما يعني أن مستوى العزل متحقق افتراضيًا (لا بيانات مشتركة بين عملاء مختلفين لأن كل عميل لديه نسخته المستقلة). لكن هذا يأتي على حساب الجهد المضاعف في الإدارة والتحديث؛ فإذا كان لدينا 100 عميل يستخدمون نظامًا تقليديًا، فهناك 100 نسخة يجب صيانتها، أما في SaaS فغالبًا هناك **نشر واحد يخدم الجميع**.

**البنية التحتية والانتشار الجغرافي:** عادةً ما تُبنى تطبيقات SaaS للاستفادة من البنية السحابية العالمية – أي يمكن نشر الخدمة عبر مراكز بيانات متعددة المناطق لضمان سرعة الاستجابة للمستخدمين أينما كانوا، وكذلك لتحقيق التوفر العالي (High Availability) والتعافي من الكوارث بسهولة. التصميم المعماري هنا **مرتكز على السحابة**، مع استخدام خدمات مثل قواعد البيانات المدارة، التخزين السحابي، وخدمات التوجيه وغيرها. بالمقابل، النظام التقليدي غالبًا يعمل في مركز بيانات محدد أو خادم محلي ضمن مقر الشركة، ما قد يحد من الوصول العالمي ويضع مسؤولية التوفر واستمرارية الخدمة على عاتق فريق تقنية المعلومات لدى العميل. إذا أرادت شركة تشغيل النظام التقليدي في عدة فروع أو دول، فعليها تكوين بنية تحتية لكل موقع أو استخدام شبكات خاصة (VPN وغيرها) لربط المستخدمين بالخادم المركزي، بينما SaaS جاهز للوصول عبر الإنترنت من أي مكان [7].

**التكامل مع الأنظمة الأخرى:** نظرًا لأن SaaS يُقدم كخدمة موحدة عبر واجهة ويب أو API، فإن التكامل مع الأنظمة الأخرى يتم عادةً من خلال **واجهات برمجية (APIs) موحدة** يقدمها مزود الخدمة. هذا يتطلب تصميمًا معماريًا يشتمل على بوابات API وآليات المصادقة متعددة المستأجرين للسماح بربط خدمة SaaS مع تطبيقات العملاء (مثل تكامل نظام محاسبي SaaS مع متجر إلكتروني SaaS آخر أو مع بوابة دفع إلكترونية). في الأنظمة التقليدية، قد يكون التكامل أكثر **تخصصًا لكل حالة** ؛ حيث قد يقوم فريق التطوير بكتابة تكامل مخصص بين النظام المثبّت محليًا ونظام آخر مستخدم لدى نفس الشركة، أو ربما يعتمد على قدرات التصدير والاستيراد اليدوي. بشكل عام، **عمارة SaaS تكون أكثر انفتاحًا وقابلية للتكامل المعياري** (بفضل استخدام واجهات HTTP/JSON المعيارية) مقارنة بالأنظمة التقليدية التي ربما تستخدم بروتوكولات أو قواعد بيانات خاصة يصعب دمجها دون تطوير إضافي.

**الخلاصة المعمارية:** تصميم عمارة SaaS يجب أن يراعي **المرونة وقابلية التوسع والأمان المشترك** منذ البداية، ويتطلب استثمارات كبيرة في تخطيط البنية السحابية وإدارة المستخدمين المتعددين. أما تصميم عمارة نظام تقليدي فيركز على **تلبية متطلبات عميل محدد بأفضل شكل ممكن ضمن بيئته الخاصة** ، مما قد يسمح بمزيد من الحرية في تخصيص البنية (من حيث اختيار أي تقنية أو هيكل دون القلق حول تعميمها) ولكنه يفتقر لمزايا المرونة على نطاق واسع التي يوفرها SaaS.

## 3. إدارة المستخدمين والبيانات وتعدد المستأجرين

**نموذج إدارة المستخدمين في SaaS:** في الأنظمة متعددة المستأجرين، يتم تصميم إدارة المستخدمين بحيث يكون هناك **فصل واضح بين نطاق كل مستأجر** . أي أن كل شركة أو عميل (مستأجر) يكون له مجموعة مستخدمين خاصة به لا يمكنهم رؤية أو الوصول لبيانات مستأجر آخر. عادةً ما يحتوي نظام SaaS على مفهوم "مدير المستأجر" أو مشرف الحساب لدى كل عميل والذي يمكنه إدارة مستخدمي شركته (إضافة/حذف مستخدمين، تحديد أدوار وصلاحيات داخل نطاق شركتهم). بالمقابل، قد يوجد أيضًا **مشرف عام** لدى مزود الخدمة يمتلك صلاحيات على مستوى جميع المستأجرين لإدارة الخدمة ككل. هذا النموذج الهرمي يضيف طبقة تعقيد إضافية مقارنة بالأنظمة التقليدية أحادية العميل، حيث يكون **جميع المستخدمين تابعين لنفس المؤسسة** ولا حاجة لتقسيمهم حسب مستأجر.

**الهوية الموحدة والتكامل مع نظم الهوية:** من الشائع في SaaS توفير خدمات دخول موحد (Single Sign-On) ودعم مزودي الهويات الخارجية لكل مستأجر. مثلًا قد تريد شركة تستخدم خدمة SaaS ربط تسجيل الدخول بـ Active Directory أو OAuth خاصتهم، مما يتطلب من نظام SaaS أن يكون قادرًا على **التعامل مع مصادر هوية متعددة** وتطبيق سياسات أمنية مختلفة لكل مستأجر. أما في النظام التقليدي المثبّت لدى الشركة، فيمكن ببساطة دمجه ضمن **نظام الهوية الداخلية مباشرة** لأنه يعمل ضمن بيئة الشركة الخاصة (مثل التكامل مع Active Directory بشكل مباشر، إذ يمتلك النظام وصولًا لبيئة الشركة الداخلية). بالتالي في SaaS هناك تحدي **تصميم نظام مرن لإدارة المستخدمين** يقبل إعدادات مختلفة لكل عميل دون تدخّل برمجي لكل حالة على حدة.

**عزل البيانات وضمان الخصوصية:** من أهم جوانب تعددية المستأجرين هو **آلية عزل البيانات** . في تطبيق SaaS واحد وقاعدة بيانات مشتركة، يجب ضمان أن استعلامات البيانات وعمليات التخزين يتم تقييدها بشكل صارم بمعرّف المستأجر. يواجه المهندسون تحديات في هذا المجال مثل منع أي تسرب للبيانات بين المستأجرين عن طريق الخطأ. يتم تحقيق ذلك عبر التصميم الجيد لطبقة الوصول إلى البيانات واعتماد مبادئ مثل النطاق *(scoping)* على مستوى كل طلب بحيث يحمل معرف المستأجر ويتم تطبيقه على جميع العمليات [8] . في حين أن الأنظمة التقليدية ليست معنية بهذه المشكلة على الإطلاق – فلكل عميل قاعدة بياناته المنفصلة أو مثيله الخاص – مما يجعل عزل البيانات أمرًا بديهيًا. لكن يجدر الذكر أنه بالرغم من سهولة العزل في النموذج التقليدي، إلا أنه قد يؤدي لصعوبات في تجميع البيانات أو الحصول على تحليلات شاملة (فمثلاً مزود SaaS يمكنه بسهولة جمع بيانات استخدام مجمّعة عبر جميع العملاء لتحسين المنتج، بينما في بيئة منفصلة لكل عميل يكون من الصعب تحقيق رؤية شاملة كهذه).

**هيكلية قاعدة البيانات متعددة المستأجرين:** هناك عدة استراتيجيات لتنظيم البيانات في تطبيقات SaaS متعددة المستأجرين [9] [6] :

- قاعدة بيانات واحدة بمخطط *(Schema)* مشترك: جميع المستأجرين يستخدمون نفس جداول البيانات مع إضافة عمود يحدد هوية المستأجر (tenant_id) في كل جدول. هذه الطريقة **أبسط من ناحية التصميم**

**والتكلفة** ولكنها تتطلب حرصًا شديدًا في كل استعلام لضمان فلترة البيانات حسب المستأجر [9] . قد تظهر تحديات عند **ارتفاع الحمل وعدد المستأجرين** بخصوص الأداء.

- قاعدة بيانات واحدة بمخططات منفصلة لكل مستأجر: تستخدم قاعدة بيانات مشتركة لكن يُعطى كل مستأجر مخططًا خاصًا به داخلها. يحقق ذلك **عزلًا أفضل للبيانات** من الطريقة الأولى (حيث لا تتشارك الجداول نفسها بين العملاء) [10] ، لكنه يزيد من تعقيد عمليات **الصيانة** مثل تطبيق التعديلات على بنية قاعدة البيانات عبر عشرات أو مئات المخططات المنفصلة.
- قواعد بيانات منفصلة تمامًا لكل مستأجر: هنا يتم إنشاء قاعدة بيانات مستقلة لكل عميل SaaS [11] . يحقق هذا النموذج **أقصى درجات العزل والأمان** (فلا شيء مشترك على مستوى التخزين بين العملاء) وهو ملائم لعملاء المؤسسات الكبرى أو القطاعات الحساسة (مثل البنوك والرعاية الصحية) التي تتطلب امتثالًا عاليًا [12] . لكن يؤخذ عليه **الكلفة المرتفعة والصعوبة التشغيلية** إذا تضخم عدد العملاء (عشرات أو مئات قواعد البيانات التي يجب إدارتها). في الواقع، هذا النموذج أقرب ما يكون لفكرة "الاستضافة المنفصلة لكل عميل" وقد يعتبر نوعًا من حلول *Single-tenant* لكن مستضاف لدى مزود الخدمة.

اختيار الاستراتيجية المناسبة يعتمد على **حجم التطبيق ونوع العملاء ومتطلبات الأمان** . فالكثير من تطبيقات SaaS الصغيرة والمتوسطة تبدأ بالنهج الأول (مشترك تمامًا) لسهولة تنفيذه ورخص تكلفته، ثم ربما تنتقل إلى مخططات منفصلة أو قواعد مستقلة لبعض العملاء الكبار الذين يستدعون متطلبات عزل أعلى.

**مزايا وعيوب تعددية المستأجرين:**

- المزايا: من منظور مزود الخدمة، يحقق نموذج تعددية المستأجرين **وفورات هائلة في الكلفة والجهد**. فبدل تشغيل بنية تحتية لكل عميل، يتم **مشاركة الموارد على نطاق واسع** مما يخفض كلفة كل عميل على حدة [13] [14] . أيضًا يسهل هذا النموذج **نشر التحديثات بانتظام للجميع** – أي تحديث يتم في بيئة مشتركة ينعكس فورًا على كل العملاء دون حاجة لترقية منفصلة لكل عميل [15] . وكذلك يجعل **عملية الإعداد والإطلاق للعملاء الجدد سلسة** ؛ فبمجرد اشتراك العميل يمكن إنشاء حساب له على المنصة خلال دقائق دون تركيب برمجيات أو ضبط خوادم [16] . المشاركة في الموارد تعني أيضًا **كفاءة أعلى في الاستفادة من العتاد** ؛ فالأوقات التي يكون فيها بعض العملاء غير نشطين قد يستفيد منها عملاء آخرون، مما **يحسّن الأداء العام** ويمنع إهدار الموارد غير المستغلة. - العيوب: في المقابل، هناك **تحديات ومخاطر** مصاحبة. أبرزها المخاطر الأمنية؛ إذ أن بيانات العملاء تكون موجودة **"خارج نطاق سيطرتهم المباشرة"** في سحابة مشتركة، مما قد يثير مخاوف بشأن الخصوصية وتسرب المعلومات [17] . أي خرق أمني في تطبيق متعدد المستأجرين قد يؤدي إلى تأثير أكبر نظرًا لإمكانية تعرض بيانات عدة شركات في آن واحد. كما أن **تشارك الموارد** قد يؤدي إلى مشكلات "جار مزعج"؛ أي إذا قام عميل ما باستهلاك موارد هائلة فجأة (كحملة تسويق إلكتروني ضخمة في متجر أحد التجار ضمن المنصة)، قد يتأثر أداء العملاء الآخرين الذين يتشاركون نفس البنية [18] إلا إذا تم تصميم النظام بذكاء لتخصيص الموارد وضمان جودة الخدمة لكل مستأجر. أضف إلى ذلك، يكون **التخصيص (Customization)** مقيدًا في بيئة SaaS متعددة المستأجرين؛ فلا يمكن تلبية طلب خاص لعميل واحد بتعديل على مستوى الكود بسهولة لأن ذلك سيؤثر على جميع العملاء الآخرين [2] . عوضًا عن ذلك يتيح مزودو SaaS قدرًا معينًا من التخصيص عبر الإعدادات والقوالب، لكن ليس بالحرية الموجودة في الحلول المخصصة لكل عميل.

**إدارة البيانات بين SaaS والتقليدي:** في النظام التقليدي المخصص لشركة واحدة، غالبًا ما تُدار البيانات داخل حدود تلك الشركة (على خوادمها أو قواعدها) مما يسهل موائمة سياسات إدارة البيانات مع سياسات الشركة الداخلية. أما في SaaS، فيلتزم مزود الخدمة بمسؤولية إدارة وحماية بيانات عشرات أو مئات العملاء وفق **معايير عالمية للأمان والخصوصية** . يشمل ذلك توفير النسخ الاحتياطية الدورية، التشفير، وآليات استرجاع البيانات عند الطلب وغيرها لجميع العملاء ككل. لذا تكون **سياسات إدارة البيانات موحدة** وتُذكر عادة في اتفاقية مستوى الخدمة (SLA) وتطبق على الجميع. في مثال **نظام محاسبي SaaS** ، يقوم المزود مثلًا بعمل نسخ احتياطية مشفرة لحسابات جميع الشركات بشكل مركزي، بينما في النظام المحاسبي التقليدي قد يتولى قسم الـ IT في كل شركة إعداد النسخ الاحتياطية الخاصة به وفق جدوله وإمكاناته.

باختصار، **يتطلب تصميم وإدارة SaaS تعددية المستأجرين فهمًا عميقًا لآليات العزل والمشاركة** لضمان استفادة الجميع من نفس المنصة بأمان وعدالة. أما **إدارة المستخدمين والبيانات في نظام تقليدي** فهي أبسط ضمن نطاق واحد، لكنها تفتقر لميزات المرونة السحابية وتتطلب موارد منفصلة لكل عميل.

## 4. آليات النشر والتحديث والصيانة

**نموذج النشر في SaaS:** في البرمجيات كخدمة، يقوم المزود **بنشر التطبيق مركزيًا في بيئة السحابة** بحيث يتمكن جميع العملاء من الوصول إليه عبر الإنترنت (عادة من خلال متصفح أو تطبيق عميل خفيف). عملية نشر إصدار جديد أو ترقية تتم **مرة واحدة على بيئة الخادم** ليصبح الإصدار الجديد متاحًا لجميع المستخدمين فورًا [19] . أي أن التحديثات **تُدار مركزيًا** بواسطة فريق مزود الخدمة. هذا الأسلوب يعني أنه يمكن إصدار تحسينات وتصحيحات أمنية بشكل مستمر وسريع (حتى بشكل يومي أو أسبوعي) دون أي تدخل من جهة العميل. العديد من مزودي SaaS يعتمدون منهجية **التسليم المستمر (Continuous Delivery)** حيث تكون عملية التحديث آلية وسلسة، ويستفيد جميع العملاء دائمًا من **آخر نسخة** من التطبيق [19] .

**نموذج النشر في الأنظمة التقليدية:** بالمقابل، **يتم نشر النظام التقليدي بشكل منفصل لكل عميل أو موقع** . فقد تقوم شركة البرمجيات بتسليم حزمة التثبيت لقسم تقنية المعلومات في الشركة العميلة ليتم تثبيتها على خوادمهم، أو ربما تتولى بنفسها تثبيت النظام في موقع العميل. عندما يصدر تحديث أو إصدار جديد، يجب **تطبيقه لكل عميل على حدة** إما بواسطة فريق العميل أو عبر زيارة/اتصال من فريق الدعم التقني. هذا يؤدي إلى واقع أن مختلف العملاء قد يكونون على **إصدارات مختلفة من البرنامج** في أي وقت، إذ قد يُحدث بعضهم إلى آخر نسخة فور صدورها بينما يتأخر آخرون لأشهر أو حتى يتخطون بعض الإصدارات. نتيجة لذلك يحدث **تجزؤ في الإصدارات** (Version Fragmentation) في قاعدة المستخدمين التقليدية [19] ؛ فنجد فريقًا ما لا يزال يعمل على إصدار قديم بينما انتقل آخرون إلى إصدار أحدث، مما يصعّب دعم المنتج. هذه المشكلة غير موجودة في SaaS لأن الجميع دوماً على النسخة ذاتها.

**السهولة والسرعة في الإعداد:** يتميز SaaS بإمكانية **التهيئة السريعة للعملاء الجدد** . فالعميل الجديد يحتاج فقط إلى إنشاء حساب وضبط إعداداته عبر واجهة الويب ليصبح جاهزًا للعمل، دون انتظار عمليات تثبيت مطوّلة. تشير التقارير إلى أن نشر حلول SaaS قد يتم **في غضون أيام** مقارنةً بأشهر في التطبيقات المؤسسية التقليدية [20] . السبب هو أن مزود SaaS يكون قد أعد البنية مسبقًا (الخوادم والبنية التحتية والخطوات الروتينية للتثبيت)، وما على العميل إلا الاشتراك للحصول على جزء من هذه الموارد المشتركة . أما في النظام التقليدي، فكل عملية تركيب جديدة تعني غالبًا مشروع نشر مستقل يشمل تجهيز الخوادم لدى العميل وضبط الشبكات وتثبيت قواعد البيانات والتطبيق...إلخ، مما يستغرق وقتًا أطول بكثير.

**التحديثات والترقيات:** كما أسلفنا، في SaaS **يتم تطبيق التحديثات تلقائيًا** لجميع المستخدمين. أي عند إضافة ميزة جديدة أو تحسين معين، سيظهر لدى جميع العملاء دون حاجة لخطوات من جانبهم [19] . هذا يضمن تجربة موحدة وحديثة للمستخدمين ويقلل بشكل كبير من عبء الصيانة على عاتقهم. في الأنظمة التقليدية، غالبًا ما يُصدر المزود حزم ترقية (patches) أو نسخ جديدة وعلى العميل تقرير **موعد وكيفية تطبيقها** . بعض المؤسسات تؤجل التحديثات خوفًا من تعطيل أنظمتها المستقرة أو لتجنب الكلفة، مما يؤدي أحيانًا إلى استخدام نسخ قديمة لا تحتوي آخر التصحيحات. مثلاً، **في برمجيات المحاسبة** نرى ذلك بوضوح: *QuickBooks Desktop* (التقليدي) يتطلب من المستخدم تنزيل التحديث وتثبيته يدويًا، بينما *QuickBooks Online* (SaaS) **يطبق الترقيات في الخلفية بشكل مستمر** بحيث يكون جميع المستخدمين على آخر إصدار دون أي جهد [21] . هذا الفارق يؤدي إلى أن **إدارة الإصدارات في SaaS أبسط** بكثير من البيئة التقليدية التي تتطلب تخطيطًا لمشاريع الترقية وصيانة توافقية الإصدارات المتعددة.

**الصيانة والدعم:** في نموذج SaaS، **يتولى المزود معظم أعمال الصيانة التقنية** : مراقبة الخوادم، النسخ الاحتياطي الدوري، إدارة قواعد البيانات، حماية الأمان، معالجة الأعطال... كل ذلك يتم مركزيًا كجزء من الخدمة المدفوعة. الشركات المستخدمة للخدمة لا تحتاج إلا إلى الاهتمام باستخدامهم الخاص (مثل إدارة حساباتهم وبياناتهم) دون الغوص في تفاصيل تقنية. هذا يتيح **تركيز العملاء على أعمالهم الأساسية** بدل الانشغال بالجوانب التقنية [22] . أما في الأنظمة التقليدية، فإن **عبء الصيانة يقع على عاتق العميل أو فريق مختص يخدمه** . يجب على قسم الـIT المحلي التأكد من صحة عمل الخوادم، أخذ نسخ احتياطية، تركيب ترقيعات الأمان لنظام التشغيل وقاعدة البيانات...إلخ. وإن كان المزود يوفر عقد دعم، فقد يساعد في بعض المهام لكن **تبقى المسؤولية النهائية محلية** . لهذه الأسباب، غالبًا ما تتطلب الحلول التقليدية وجود **فريق تقني داخلي** لدى الشركة يمتلك خبرة في تشغيل وصيانة النظام، بينما يمكن تبني SaaS حتى من قبل شركات صغيرة دون فريق تقني كبير [23] .

**التوفر واستمرارية الخدمة:** يلتزم مزودو SaaS عادةً باتفاقيات مستوى خدمة (SLA) تضمن **نسبة تشغيل عالية (High Uptime)** تتجاوز 99% مثلا، حيث يستخدمون بنى تحتية زائدة عن الحاجة (ريداندنسي) وأنظمة مراقبة على مدار الساعة لتحقيق ذلك. عند الحاجة إلى أعمال صيانة دورية، يتم جدولة ذلك في أضيق الحدود وخارج أوقات الذروة وبشكل يؤثر على جميع العملاء معًا لفترة وجيزة. في المقابل، الأنظمة التقليدية يعتمد توفرها على البنية المحلية لكل عميل؛ فإن لم يكن لدى الشركة بنية احتياطية فقد تواجه توقف الخدمة أثناء أعطال الخادم أو الصيانة. ولكن من جهة أخرى، يعطي النظام التقليدي **مرونة للعميل في جدولة التوقفات** (مثل اختيار وقت الترقية بما يناسبه)، بينما في SaaS قد يفرض المزود نافذة صيانة عامة تطال كل المستخدمين في وقت محدد.

**خلاصة النشر والصيانة:** يتميز SaaS **بالسرعة والسهولة في النشر والتحديثات المستمرة** التي لا تتطلب تدخل العملاء، مما يوفر عليهم الجهد ويضمن امتلاكهم لأحدث الميزات دائمًا [19] . بالمقابل، تكون **الأنظمة التقليدية أكثر مرونة في التحكم من قبل العميل** (فيقرر متى يُحدّث أو يبقى على إصدار معين) لكنها تتطلب موارد وجهود صيانة أعلى ومستمرة من طرفه. الاتجاه العام في الصناعة هو تفضيل النموذج السحابي لصيانة أقل وتحديثات أسرع، إلا في الحالات التي تفرض بقاء النظام تحت إدارة العميل لأسباب خاصة.

## 5. أمثلة واقعية: SaaS مقابل الأنظمة التقليدية في المجال

لتوضيح الفروقات آنفة الذكر، فيما يلي **حالات واقعية لشركات ومنتجات** عملت في نماذج SaaS ومقابلها نماذج تقليدية، خصوصًا في نطاق **المحاسبة** و**التجارة الإلكترونية متعددة التجار**:

- **نظام المحاسبة Intuit QuickBooks:** يعتبر QuickBooks Desktop أحد أشهر المنتجات التقليدية لإدارة المحاسبة للشركات الصغيرة، حيث يُثبّت محليًا على حاسوب أو خادم العميل. على الجانب الآخر، قدمت Intuit خدمة **QuickBooks Online** (QBO) كنموذج SaaS لتلبية حاجة العملاء للحوسبة السحابية. نرى بوضوح الفوارق بين الإصدارين: **QBO متاح من أي مكان عبر الإنترنت مع قدرة لعدة مستخدمين على العمل سويًا في نفس البيانات بشكل لحظي** ، مما يجعله ملائمًا لفِرق المحاسبة التي تحتاج تعاون آني [24] [25] . كما يتكامل QBO بسهولة مع خدمات أخرى عبر الإنترنت ويوفر واجهات حديثة سهلة الاستخدام، في حين أن **QuickBooks Desktop يعمل فقط على الأجهزة المثبت عليها** (لا يمكن الوصول إليه عن بعد إلا بإعدادات شبكية خاصة) ويقدم مزايا أكثر تقدمًا للمستخدم الفردي لكن دون نفس القدرات التعاونية [24] . من ناحية **التحديثات** ، تقوم Intuit بتحديث QuickBooks Online باستمرار في الخلفية، بينما إصدارات QuickBooks Desktop تتلقى ترقيات سنوية أو تحديثات يدوية يجب على المستخدم تطبيقها [21] . حتى من جانب **التسعير** ، QBO يعتمد اشتراكًا شهريًا مع خطط تتراوح من \$35 إلى \$100 شهريًا حسب الميزات، أما QuickBooks Desktop فيُشترى بتراخيص سنوية وصلت إلى آلاف الدولارات لبعض الإصدارات المتقدمة [26] [27] . هذه الفروق ترجمت إلى واقع أن QBO خيار مفضل لمن يريد المرونة والتعاون، بينما يظل Desktop مناسبًا لمن يهتم بالتحكم الكامل والميزات المتقدمة في بيئة منعزلة.

- **منصات التجارة الإلكترونية: Shopify مقابل Magento (Adobe Commerce):** Shopify منصة SaaS معروفة عالميًا تمكن الشركات والأفراد من إنشاء متجر إلكتروني كامل الوظائف بسرعة وسهولة عبر واجهة ويب، دون حاجة لإدارة الخوادم أو البرمجة. في المقابل، Magento (الذي تطور لاحقًا إلى Adobe Commerce) هو منصة تجارة إلكترونية تقليدية مفتوحة المصدر **يتم تثبيتها وإدارتها من قبل التاجر أو مستضيف طرف ثالث** . شهد هذا المجال تحولًا كبيرًا نحو SaaS في السنوات الأخيرة؛ **فقد نمت حصة المنصات السحابية مثل Shopify بشكل هائل لتتجاوز حصة الأنظمة التقليدية** كماغنتو بمقدار الضِّعف تقريبًا [28] . يكمن السبب في أن SaaS مثل Shopify يقدم تجربة **جاهزة out-of-the-box** تشمل الاستضافة والتأمين والتحديث المستمر وواجهة إدارة بسيطة يمكن لغير التقنيين التعامل معها [29] . يمكن إطلاق متجر عبر Shopify في أيام قليلة مع تكاليف بدء تشغيل منخفضة (اشتراك شهري يبدأ من عشرات الدولارات) [30] ، بينما إطلاق متجر *Magento* يتطلب فريق تطوير لإعداد الاستضافة وتخصيص المتجر وقد تصل تكلفة الترخيص والدعم لمئات الآلاف من الدولارات سنويًا في حالة النسخة المدفوعة للمؤسسات [31] . أحد الأمثلة، شركة *Crate & Barrel* ومتجر *Steve Madden* وهما علامتان تجاريتان انتقلتا لاستخدام Shopify Plus (نسخة SaaS للمؤسسات) لتلبية احتياجات التجارة الإلكترونية الخاصة بهما [28] ، في حين أن شركات أخرى ذات متطلبات تخصيص عميقة جدًا فضلت الاستمرار على Magento لمنحها الحرية في تعديل كل جوانب المنصة. من حيث **الميزات** ، يوفر النموذجان معظم وظائف التجارة (كت katalog المنتجات، إدارة

المخزون، العروض الترويجية)، ولكن SaaS يكسب في **سرعة الحصول على المزايا الجديدة** (حيث يضيفها المزود مركزيًا لجميع العملاء) بينما يكسب النموذج التقليدي في **قابلية التخصيص اللامحدودة** عبر الوصول للكود المصدري. اليوم، حتى Magento قدّم عروض SaaS (مثل Magento Commerce Cloud) لتلبية الطلب، مما يعكس التقارب نحو نموذج الخدمات المستضافة.

- **حلول تخطيط الموارد (ERP) – أمثلة متنوعة:** الكثير من بائعي الأنظمة الضخمة التي كانت تقليدية تاريخيًا قدموا نُسخ SaaS من منتجاتهم. مثلًا **شركة SAP** المعروفة بأنظمتها الواسعة الانتشار أطلقت منتج *SAP S/4HANA Cloud* كإصدار SaaS من نظام ERP الخاص بها، إلى جانب استمرار توفير النسخة المحلية *S/4HANA On-Premise*. كذلك شركة **Oracle** لديها *Oracle NetSuite* كنظام ERP/MRP سحابي متعدد المستأجرين اكتسب شعبية واسعة، مقابل حلول **Oracle E-Business Suite** القديمة المثبتة محليًا. حتى على مستوى المصادر المفتوحة، **منصة Odoo** (وهي نظام ERP/CRM) توفر خيار SaaS مستضاف على سيرفرات الشركة للأعمال الصغيرة التي لا تريد إدارة البنية، إضافة لإمكانية تنزيل البرنامج وتشغيله على خوادم العميل لمن يفضل ذلك. هذه الأمثلة توضح اتجاه السوق نحو **ازدواجية الخيارات** – فالشركات البرمجية تعرض لعملائها الخيار بين SaaS سريع منخفض التكلفة الأولية، أو نسخة تقليدية لمن يحتاج التحكم الكامل. والنتيجة أن العديد من المؤسسات الجديدة باتت تميل إلى **تبني SaaS للتطبيقات الداعمة للأعمال** (كالمحاسبة والموارد البشرية والتسويق) لسرعة ما يوفره من قيمة، بينما تحتفظ بالتطبيقات المحلية فقط حين يكون هناك مبرر قوي يتعلق بالأمان أو التخصيص العميق.

- **أمثلة أخرى:** تطبيقات التعاون والعمل الجماعي شهدت أيضًا هذا التحول. فمثلًا **Slack** كمنصة مراسلة وتعاون هي نموذج SaaS نجح عالميًا مستبدلًا حلولًا تقليدية سابقة (مثل خوادم الدردشة الداخلية أو حتى الاعتماد على البريد الإلكتروني للتنسيق). أيضًا في مجال إدارة علاقات العملاء CRM لدينا **Salesforce** الذي كان من أوائل حلول SaaS الناجحة تجاريًا في مواجهة حلول CRM تقليدية كـSiebel Systems. في مجال المحتوى وإدارة الوثائق، نرى **Google Workspace** و**Microsoft 365** كخدمات SaaS مقابل حزم *Microsoft Office* وSharePoint التقليدية – علماً أن مايكروسوفت نفسها انتقلت لهذا النموذج الهجين بتقديم خدمات Office 365 وDynamics 365 على السحابة إلى جانب نسخ الخوادم التقليدية.

هذه الحالات الواقعية تبين أنه **لا يوجد حل واحد مناسب لكل ظرف**؛ فبعض المؤسسات جرّبت SaaS ولم يناسبها بسبب قيود معينة ففضلت التقليدي، وأخرى تخلت عن أنظمتها القديمة لتواكب العصر عبر SaaS. لكن الاتجاه عمومًا يشير إلى نمو متسارع في تبني SaaS عبر مختلف القطاعات [32] نظرًا لما يقدمه من سهولة وتكلفة موزعة على الاستخدام [22]. الشركات الحديثة النشأة خاصةً تفضل حلول SaaS لتجنب الاستثمار الكبير مقدمًا والتركيز على جوهر عملها دون بناء قسم تقنية معلومات متكامل [14].

## 6. الأخطاء الشائعة في تصميم وتنفيذ أنظمة SaaS وكيفية تجنبها

على الرغم من مزايا نموذج SaaS، هناك مجموعة من **الأخطاء الشائعة التي يقع فيها الفرق عند تصميم وتطوير تطبيقات SaaS** قد تؤدي إلى مشكلات كبيرة في قابلية التوسع أو العمليات لاحقًا. فيما يلي أبرز هذه الأخطاء (وفق خبراء المجال) وكيفية تلافيها:

- **عدم التخطيط للتعددية منذ البداية:** من أكثر الأخطاء جوهرية هو البدء بتطوير التطبيق كسطح أحادي المستأجر (Single-tenant) موجه لعميل واحد ثم محاولة تحويله لاحقًا إلى SaaS متعدد المستأجرين. جون توبّر (خبير سحابي) أشار إلى أن كثيرًا من منتجات SaaS الناجحة بدأت أصلًا كمشاريع موجهة لمستأجر منفرد، لكن **إهمال اعتبار جوانب التعددية من اليوم الأول يمكن أن يعيق بشدة قدرة المنتج على التوسع لاحقًا** [3]. الحل هو **دمج مفهوم المستأجر في جميع مكونات التطبيق منذ البداية**: في تصميم قاعدة البيانات، طبقات التطبيق، واجهات البرمجة... بحيث يصبح توسيع التطبيق لخدمة عملاء متعددين مسألة تكوين وإعداد بدلاً من إعادة هندسة جذرية.

- **تخصيصات العملاء المفرطة:** قد يضغط بعض العملاء الكبار للحصول على ميزات خاصة بهم أو تعديلات استثنائية في التطبيق. الرضوخ لكل تلك الطلبات وتحويل المنتج إلى نسخ مخصصة لكل عميل هو وصفة كارثية لقتل نموذج *SaaS*، لأنه يزيد التعقيد التشغيلي بشكل هائل. يؤكد الخبراء أن **تقديم ميزات فريدة**

**أو إصدارات مختلفة لكل مستأجر سيُنهك فرق العمليات ويهدم وفورات schaal** [33] . الحل الأمثل هو التمسك بفلسفة منتج قوية تقول "لا إستراتيجي" لبعض الطلبات الخارجة عن نطاق رؤية المنتج [33] . يجب إقناع العملاء أن قوة SaaS في كونه منصة موحدة، وأن التخصيص يكون عبر الإعدادات وليس بتفريع الكود لكل عميل. هذا النهج يتطلب شجاعة من فريق المنتج، لكنه ضروري للحفاظ على الاستدامة – يمكن تقديم بدائل مثل بناء الميزة إذا كانت ستفيد شريحة كبيرة من العملاء لا عميل واحد فقط.

- **عدم أتمتة تهيئة المستأجر الجديد:** مع نمو قاعدة عملاء SaaS، قد يصبح إعداد بيئة لكل عميل جديد عبئًا إذا تم يدويًا. أحد الأخطاء هو الاعتماد على تدخل المطورين في كل مرة لإعداد عميل جديد أو بيئة تجريبية. ينصح الخبراء بأتمتة عملية توفير المستأجر الجديد *(Tenant Provisioning)* مبكرًا قدر الإمكان [34] . وجود سكربتات أو أدوات تنشئ حسابات العملاء الجديدة، وضبط مساحاتهم الخاصة، والربط مع النطاق (domain) الخاص بهم أو غيره تلقائيًا سيمنع عنق الزجاجة لاحقًا عندما تتوالى عمليات الاشتراك. تجنب هذا الخطأ يضمن أن فريق التطوير لن ينشغل بطلبات التهيئة بدل التركيز على تحسين المنتج.

- **عدم حساب تكلفة الخدمة لكل مستأجر بدقة:** قد يقع بعض مؤسسي SaaS في خطأ تسعير الخدمة دون فهم التكلفة الحقيقة لخدمة كل عميل. ينصح توبّر بحساب **الحد الأدنى للتكلفة لكل مستأجر** (موارد الخادم، الخزن، التراخيص) منذ البداية والتأكد من أن نموذج التسعير يغطي هذه التكلفة margin مقبول [3] . عدم القيام بذلك قد يوقع الشركة في خسائر تصاعدية كلما زاد العملاء بدلاً من أرباح، أو يضطرها لاحقًا لرفع الأسعار بشكل يزعج العملاء. لذا التخطيط المالي جزء مهم من التصميم الفني – اختيار نموذج متعدد المستأجرين فعال واستغلالي للموارد يساعد في خفض تكلفة الوحدة الواحدة.

- **نشر أجزاء من الحل في بيئة العميل (كحلول هجينة خاطئة):** أحيانًا يطلب بعض العملاء لأسباب تتعلق بالسيادة على البيانات أو التكامل، أن يتم **نشر جزء من تطبيق SaaS داخل بنيتهم الخاصة** . إذا تم الاستجابة بلا حذر، سنجد الفريق نفسه يدير **حلولًا هجينة** تجمع بين SaaS والاستضافة المخصصة مما يزيل مزايا كلا النموذجين. بالفعل يؤكد الخبراء: "إذا سمحت للعملاء أن يملوا أين ومتى تحدث التحديثات، فإنك تضيف تعقيدًا عمليًا جسيمًا وتضحي بعطل نهاية الأسبوع الخاص بفريقك" [35] . وبالمثل، **نشر البرنامج في حساب AWS الخاص بالعميل ليس حقًا SaaS** بل أقرب لبيع البرمجيات التقليدي [36] . لتجنب ذلك، يجب تثقيف العميل منذ البداية على حدود نموذج SaaS، واقتراح حلول بديلة مثل توفير واجهات API أو بوابات تمكن التكامل دون الحاجة لاستضافة فرعية. الحفاظ على **نواة التطبيق في بيئة موحدة** هو ما يجعل SaaS ممكنًا، وأي استثناءات يجب أن تكون محدودة جدًا ومدروسة بعناية (مثلاً تقديم خيار **منصة خاصة Private Cloud** لعملاء محددين جدًا يدفعون تكاليف أعلى، إن كان لا بد من ذلك).

- **إهمال جوانب النسخ الاحتياطي والطوارئ:** خطأ آخر خطير هو **افتراض أن البنية السحابية لن تفشل** أو تأجيل إعداد خطط التعافي من الكوارث (Disaster Recovery). يذكر Werner Vogels (CTO في أمازون) مقولة شهيرة: "كل شيء يفشل طوال الوقت" ، وينصح توبّر بأن يتبنى فريق SaaS هذا المبدأ عبر **وضع خطة مسبقة للتعامل مع الفشل** – إجراء "ما قبل الوفاة" (Pre-mortem) لتخيل ما الذي قد يحدث من سيناريوهات كارثية وكيفية التعامل معها [37] . لقد شهدنا حالات فعلية لمنصات SaaS تعطلت تمامًا لساعات أو أيام بسبب غياب خطط للطوارئ (مثل أعطال قواعد البيانات دون وجود نسخ احتياطية حديثة، أو انقطاع منطقة كاملة في مزود سحابي بدون قابلية نقل التشغيل إلى منطقة أخرى). **تجنب هذا الخطأ** يكون ببناء بنية توفر عالي (مثل النشر عبر عدة مناطق جغرافية)، وإعداد نسخ احتياطية متعددة المستويات (لحظية ويومية وأسبوعية)، وتوثيق خطط استعادة الخدمة واختبارها دوريًا.

- **ضعف التخطيط الأمني وتصنيف البيانات:** قد يركز الفريق الناشئ على بناء الميزات بسرعة لجذب العملاء ويتجاهل بناء **نموذج أمني شامل** منذ البدء. من الأخطاء مثلاً عدم **تصنيف البيانات** وتحديد من له صلاحية الوصول لأي نوع منها [37] ، أو عدم تطبيق مبدأ الأقل امتيازًا في التصميم. كذلك إهمال تأمين البيئات (مثل نسيان إعداد جدار حماية أو IAM سليم) قد يترك التطبيق عرضة للاختراقات. تجنب ذلك يكون باعتماد ممارسات DevSecOps مبكرًا – أي دمج اختبارات الأمان في خط التجميع، وإجراء تدقيق أمني دوري، وتشفير البيانات الحساسة وهي في الراحة وأثناء النقل، ووضع سياسات مراقبة وتنبيه لأي نشاط مشبوه.

- **عدم فهم احتياجات العملاء الحقيقية:** من الأخطاء العامة، المشتركة مع أي منتج، هو **بناء شيء لا يريده المستخدمون** حقًا أو حل مشكلة خاطئة. قد تنجرف الفرق التقنية نحو التقنية الحديثة والإمكانيات الهائلة لسحابتهم وتنسى قضاء الوقت مع المستخدمين لفهم ألمهم الفعلي. النتيجة منتج SaaS غني بالميزات لكن قليل الفائدة [38] . يجب تجنب ذلك عبر منهجية تطوير متمحورة حول المستخدم: إجراء مقابلات، اختبارات قابلية الاستخدام، تحليل بيانات استخدام الميزات الموجودة، لضمان أن البناء في الاتجاه الصحيح.

هذه بعض أهم الأخطاء، وتجنبها يساهم في نجاح أي مشروع SaaS. بشكل عام، **يتطلب تطوير SaaS عقلية مختلفة عن البرمجيات التقليدية** : عقلية توازن بين السرعة في طرح الميزات وبين الحفاظ على اتساق المنصة لجميع العملاء، والتخطيط طويل الأمد للتوسع والأمان بدل الحلول السريعة لكل عميل على حدة.

## 7. التحديات الأمنية والامتثال: SaaS مقارنة بالأنظمة التقليدية

**نطاق المسؤولية الأمنية:** يختلف نموذج الأمان بين SaaS والتقليدي من حيث من يتحمل مسؤولية ماذا. في الحلول التقليدية (داخلية)، تقع المسؤولية الكاملة تقريبًا على عاتق قسم تقنية المعلومات لدى العميل؛ فهو مسؤول عن تأمين الخوادم الفيزيائية (حماية مبنى ومعدات)، الشبكة الداخلية، ضبط الجدار الناري، إدارة وصول المستخدمين، وتطبيق سياسات الأمن على التطبيق والبيانات. في المقابل، في نموذج SaaS هناك **مسؤولية مشتركة** : يتولى مزود الخدمة تأمين البنية التحتية السحابية والتطبيق نفسه (تصحيحات أمنية، تشفير البيانات، منع الهجمات الإلكترونية على مستوى الخادم) [39] [40] ، بينما يكون على العميل التأكد من إدارة مستخدميه وصلاحياتهم بشكل صحيح واستخدام الخدمة وفق سياسات أمن كلمات المرور وغيرها المقدمة. هذا التوزيع يخفف العبء عن العميل لكنه يتطلب **ثقة عالية بمزود الخدمة** وقدرته على الوفاء بمتطلبات الأمان.

**التحكم بالبيانات وخصوصيتها:** كثيرًا ما يُطرح تساؤل: أي النموذجين أكثر أمنًا؟ من منظور التحكم، يعطي النظام التقليدي الشركة **سيطرة كاملة على بياناتها ومواردها** ؛ فالبيانات مخزنة في قواعد داخل الشركة أو في خوادم تحت إدارتها، مما يمنح إحساسًا بالأمان لكونها تعلم أين توجد بياناتها ومن يمكنه الوصول فعليًا للخوادم [40] [41] . بينما في SaaS، البيانات عادةً **مستضافة لدى طرف ثالث** وربما موزعة عبر مراكز بيانات حول العالم. يشعر بعض العملاء بالقلق لكونهم لا يمتلكون السيطرة المادية – إذ يجب أن يثقوا بأن مزود SaaS لن يطّلع على بياناتهم إلا ضمن حدود المتفق عليها، وبأنه يوفر العزل الكافي بحيث لا يستطيع عملاء آخرون أو حتى مخترقون من الوصول إليها [17] . أيضًا مسألة **حقوق البيانات** تظهر هنا؛ فعلى العميل فهم من يملك مفاتيح التشفير وكيف يمكنه استعادة بياناته من الخدمة لو رغب في المغادرة.

مع ذلك، تجدر الإشارة إلى أنه **في كثير من الحالات تكون حلول SaaS أكثر أمانًا من الأنظمة التقليدية الداخلية** ، خاصة للشركات الصغيرة والمتوسطة [23] . ذلك لأن مزودي الخدمات السحابية يستثمرون بكثافة في أحدث ما توصل إليه مجال الأمن السيبراني (جدران نارية متقدمة، أنظمة كشف التسلل، تشفير شامل، فرق مختصة تراقب 24/7) وهي أمور قد لا تتوفر لتلك الشركات لو أدارت أنظمتها بنفسها [23] . بعبارة أخرى، انتقال العبء الأمني إلى متخصصين قد يرفع المستوى الأمني العام. كثير من مزودي SaaS اليوم يحصلون على **شهادات امتثال أمنية** صارمة (مثل ISO 27001, SOC 2) لإثبات التزامهم بمعايير الأمن الدولية، مما يعطي الثقة للعملاء.

**الفروقات في التهديدات والهجمات:** يواجه النظامان تهديدات متشابهة من حيث أنواع الهجمات (كـSQL Injection، أو هجمات حجب الخدمة DDoS، أو الهندسة الاجتماعية لاستهداف حسابات المستخدمين). إنما **عواقب الاختراق** تختلف في النموذجين: اختراق نظام تقليدي لدى شركة يعني انكشاف بيانات تلك الشركة وحدها، بينما اختراق تطبيق SaaS مركزي قد يعني تعريض بيانات عدة شركات لخطر التسريب دفعة واحدة [17] . لذلك يكون التركيز في SaaS أكبر على **توفير عزلة صارمة** بين بيانات العملاء كما ذكرنا، وأيضًا على **الاختبارات الأمنية المستمرة** لكل تحديث قبل طرحه (لأن ثغرة واحدة قد تؤثر على الجميع). ومن زاوية أخرى، العميل في النظام التقليدي قد يهمل بعض التحديثات الأمنية مما يجعله عرضة للهجمات، أما في SaaS فالمزود يحرص على تطبيق التصحيحات أولًا بأول مما يقلل تعرض الجميع لثغرات معروفة [19] .

**التحديات في الامتثال التنظيمي:** من أكثر الاعتبارات حساسية **الامتثال للقوانين والمعايير** عند استخدام SaaS مقارنة بحلول داخلية. على سبيل المثال، **اللوائح لحماية البيانات الشخصية (مثل GDPR في أوروبا)** تفرض ضوابط على مكان تخزين البيانات وكيفية استخدامها. في النظام التقليدي، قد تختار الشركة ببساطة إبقاء بيانات

عملائها الأوروبيين على خوادم ضمن أوروبا للامتثال، ولديها تحكم مباشر بذلك. أما عند استخدام SaaS عالمي، على الشركة التأكد من أن مزود الخدمة يوفر خيارات **تحديد موقع تخزين البيانات** (Data Residency) بما يتوافق مع القوانين، وأنه يوقع اتفاقيات معالجة بيانات (DPA) تضمن التزامات قانونية لحماية الخصوصية. بعض الهيئات التنظيمية في القطاعات المالية أو الصحية قد **لا تسمح بوضع بيانات حساسة على سحابة عامة مشتركة** ما لم يتم استيفاء شروط صارمة. لذلك نرى توجهًا لدى مزودي SaaS لتقديم خدمات سحابية خاصة للصناعات الحساسة أو على الأقل خيار *Single-tenant SaaS* لهؤلاء العملاء لضمان عزل بياناتهم على بنى خاصة [42] . مثلًا، **شركات الرعاية الصحية الأمريكية** ملزمة بـHIPAA، وللامتثال في نموذج SaaS يجب أن يوقع المزود اتفاقية شراكة أعمال *(BAA)* ويلتزم بضوابط HIPAA، أو يلجأ البعض للحلول المخصصة حتى لو كانت أقل كفاءة لضمان امتثال كامل [42] . في المقابل، الحلول التقليدية تتيح للشركات تطبيق إجراءات الامتثال داخل جدرانها بطريقة تراها مناسبة ولكن تتطلب منها بذل هذا الجهد بنفسها.

**المرونة في الأمان:** أحد الاعتبارات أن بعض المؤسسات تفضّل النظام التقليدي لأنها تستطيع **تطبيق سياسات أمنية مخصصة** للغاية تناسب بيئتها (مثل تشفير داخلي إضافي، أو منع النظام من الاتصال بالإنترنت تمامًا لأقصى حماية). في SaaS لا يملك العميل حرية تعديل سياسات الأمان كما يشاء – بل يعتمد على ما يقدمه المزود والذي يكون عامًا على الجميع. فإذا كان لدى شركة مثلاً متطلبات فحص أمني لكل طلب أو سجل بسبب طبيعة بياناتها، قد لا تستطيع فرض ذلك في خدمة SaaS خارجية. لهذا السبب، **القطاعات شديدة التنظيم** (كالدفاع مثلاً) ما زالت تفضل حلولًا داخلية.

**التدقيق وإدارة الوصول:** في بيئة تقليدية، يمكن للشركة استخدام أدواتها الخاصة لتدقيق كل عملية على المستوى الشبكي أو تطبيق حلول IAM خاصة. أما في SaaS، فعليها الثقة بتقارير التدقيق التي يوفرها المزود والتي تغطي أنشطة مستخدميها فقط وليس النظام ككل. بعض مزودي SaaS يقدمون ميزات متقدمة كـ**سجلات تدقيق مفصلة** لكل حدث متعلق ببيانات العميل، و **خيارات مصادقة متعددة العوامل** (2FA) وحتى **قدرات تشفير جانبية** (Client-side encryption) حيث لا يحتفظ المزود بمفاتيح التشفير. هذه الأمور ينبغي على العميل النظر فيها عند تقييم أمن أي خدمة SaaS – أي مدى شفافية المزود واستعداده لتلبية متطلبات الامتثال المحددة لصنعته.

**مقارنة موجزة بين النموذجين من منظور الأمن والامتثال:** يمكن تلخيص الفروق كالتالي:

- التحكم: يمنح النظام التقليدي تحكمًا مطلقًا بالبيانات والبنية، بينما SaaS يضع جزءًا كبيرًا من التحكم بيد المزود [40] .
- الخبرة: يتطلب التقليدي وجود خبرات أمنية داخلية قوية وإجراءات محلية، أما SaaS فيستفيد العميل من خبرات المزود الذي غالبًا ما يتبع معايير أمان صناعية عالمية [39] .
- المخاطر: المخاطر في التقليدي محلية – اختراقه يؤثر على شركة واحدة – ولكنه أيضًا أقل استهدافًا من المخترقين الكبار. SaaS قد يكون هدفًا مغريًا للقراصنة نظرًا لمركزيته وكونه "مصدر غني" بالمعلومات المتنوعة، لذا يستقطب محاولات هجومية أكثر، لكنه أيضًا عادةً مجهز بدفاعات أقوى.
- الامتثال: في التقليدي، **مسار الامتثال أبسط** لأن البيانات لا تغادر نطاق الشركة (مثلاً أسهل القول "نحن لا ننقل بيانات المرضى خارج المستشفى"). في SaaS يجب العمل مع المزود لضمان الامتثال – مثلاً توفر شهادات مثل HIPAA compliance، GDPR compliance وغيرها – وقد تكون هناك عوائق إذا اشترط القانون سيطرة مباشرة على البيانات.
- الشفافية: الشركات أحيانًا تتردد في SaaS لغياب الشفافية الكاملة؛ أي أنهم لا يستطيعون رؤية طبقات الأمن خلف الكواليس كما يفعلون في نظامهم الخاص. للتغلب على ذلك، صار المزودون يقدمون المزيد من المعلومات، كتقارير تدقيق مستقلة وشهادات أطراف ثالثة، لإثبات مستوى أمنهم.

في المحصلة، **يمكن لأنظمة SaaS تحقيق مستويات أمن وامتثال تضاهي أو تفوق الأنظمة التقليدية** إذا تم تصميمها وتشغيلها وفق أفضل الممارسات [43] . والتوجه الحديث يرى الكثير من المؤسسات الكبيرة والصغيرة على حد سواء أصبحت أكثر اطمئنانًا لوضع بياناتها في السحابة بعدما اتضح أن مزودي الخدمات يستثمرون موارد ضخمة في الأمن ربما تفوق ما تستطيع هي استثماره [23] . ومع ذلك، سيستمر وجود حالات استخدام تتطلب حلولًا تقليدية أو خاصة حفاظًا على السيطرة الكاملة، خاصةً في البيئات الحرجة. لذا يعتمد الاختيار على **طبيعة البيانات** وحساسية المجال إلى جانب عوامل أخرى كالتكلفة والراحة.

## 8. التأثير على التكلفة وقابلية التوسع وتجربة المستخدم

أحد الأسئلة الحاسمة لأي مؤسسة عند اختيار بين SaaS أو نظام تقليدي هو: كيف ستؤثر هذه البنية على التكاليف على المدى القريب والبعيد، على قدرتنا على التوسع والنمو، وعلى تجربة المستخدم النهائي؟ فيما يلي مقارنة واضحة لهذه الجوانب:

مخطط يوضح الفرق في هيكل التكاليف بين نموذج *SaaS* والنموذج التقليدي: بالنسبة للأنظمة الداخلية *(التقليدية)* هناك تكاليف ظاهرة كالتراخيص المبدئية، لكن أيضًا تكاليف خفية كبيرة مثل الأجهزة والصيانة والتحديثات والدعم الأمني [44] [45] . في نموذج *SaaS*، التكاليف الأولية أقل بكثير *(اشتراك دوري)* مع بعض التكاليف التشغيلية كالإعداد والتدريب، بينما يقوم المزود بتحمل أعباء البنية التحتية والصيانة.

### التكلفة (Cost)

**نموذج التسعير – نفقات رأسمالية vs تشغيلية:** في الأنظمة التقليدية، غالبًا ما يتم **شراء رخصة دائمة أو سنوية** للبرنامج بدفع مبلغ كبير مقدمًا، بالإضافة إلى ما يستتبعه ذلك من شراء خوادم أو تجهيز بنية تحتية لتشغيله. تعتبر هذه **نفقات رأسمالية (CapEx)** تتحملها الشركة في البداية [46] . كما قد تكون هناك **رسوم صيانة دورية** (20% مثلًا من قيمة الترخيص سنويًا) للحصول على دعم التحديثات من البائع. بالمقابل، يتبع SaaS نموذج **الاشتراك الشهري/السنوي** حيث تدفع الشركة تكلفة ثابتة أو حسب الاستخدام بشكل دوري مقابل الخدمة [46] . هذا يحوّل الإنفاق إلى **نفقات تشغيلية (OpEx)** موزعة بمرور الوقت بدلًا من مبلغ ضخم في البداية. يسهّل ذلك على الشركات وضع الميزانية والتنبؤ بالتكاليف، ويخفض حاجز الدخول أمام الشركات الصغيرة التي لا تستطيع توفير استثمار ضخم upfront [22] [47] . على سبيل المثال، بدلاً من شراء نظام بمئات آلاف الدولارات، يمكن الاشتراك بـSaaS بمئات الدولارات شهريًا فقط.

**التكلفة الإجمالية للملكية (TCO):** رغم أن SaaS يبدو أرخص في البداية، ينبغي مقارنة **التكلفة طويلة الأجل**. في بعض الحالات، إذا استُخدم النظام لسنوات عديدة فقد تتجاوز رسوم الاشتراك التراكمي ما كان سيدفع لشراء حل داخلي لمرة واحدة [47] [48] . على سبيل المثال، إذا كان ترخيص النظام التقليدي يكلف 50 ألف دولار لمرة واحدة وتكاليف الخادم والصيانة 10 آلاف سنويًا، وفي المقابل SaaS يكلف 2000 دولار شهريًا، فعلى مدى 5 سنوات ستكون الكلفة الإجمالية متقاربة. مع ذلك، غالبًا ما يُوازن العملاء ذلك مقابل المنافع الأخرى (المرونة والتحديثات) التي يحصلون عليها في SaaS. أيضًا كثير من مزودي SaaS يقدمون حزم خصم عند الدفع السنوي أو لمدد طويلة.

**التكاليف المخفية في النموذج التقليدي:** عند حساب التكلفة، يوجد **بنود خفية** في الحلول التقليدية كثيرًا ما تُغفل: كلفة **الأجهزة** (خوادم، وحدات تخزين، معدات شبكية) [49] [50] ، تكلفة **الطاقة والتبريد** لمراكز البيانات الداخلية، تكلفة **العمالة التقنية** اللازمة لإدارة النظام (رواتب المسؤولين والتدريب) [51] [50] ، كلفة **التوقفات** غير المخطط لها. بينما في SaaS، هذه البنود ضمنية ضمن اشتراك الخدمة ويتحملها المزود. الرسم أعلاه يبين كيف تكون جبال من التكاليف غير المباشرة مصاحبة للأنظمة التقليدية تحت سطح الماء، في حين أن SaaS يقلل معظمها وينقلها للمزود.

**مرونة التكاليف مع حجم العمل:** في SaaS عادةً ما تكون **التكاليف مرنة وفق حجم الاستخدام**. فمثلًا، قد يعتمد الاشتراك على عدد المستخدمين أو حجم البيانات أو عدد المعاملات. هذا يعني أنه **يمكن للشركة البدء صغيرًا بتكلفة منخفضة** ثم تزيد الدفع فقط عند نمو أعمالها أو زيادة حاجتها [52] . بل إن بعض النماذج تمكن من تقليل الكلفة عند فترات الكساد (مثال: منصة تأجير تدفع أقل في غير موسمها لانخفاض المستخدمين) [53] . أما في النظام التقليدي، فالتكلفة مقدمًا كبيرة بغض النظر عن معدل الاستخدام اللاحق، وإن حصل نمو كبير فقد تضطر الشركة لاستثمارات قفزة أخرى (شراء خادم إضافي أو تراخيص جديدة) بشكل فجائي وكبير [54] . من ناحية أخرى، إذا قلّ الاستخدام في النظام الداخلي فلن توفر الكثير حيث أن الاستثمارات تمت بالفعل. هذه **الديناميكية تجعل SaaS جذابًا من منظور الجدوى الاقتصادية** ، لأنه أشبه بتحويل الإنفاق إلى نموذج "ادفع حسب الاستعمال" الذي يتماشى مع إيرادات الشركة بشكل أفضل.

**مثال توضيحي (Shopify vs Magento):** للتوضيح العملي، عند إطلاق متجر إلكتروني على Shopify، قد تبدأ الخطة بـ$29\ شهريًا + نسبة من المبيعات [30] . إذا كان المتجر ناشئًا بمبيعات محدودة، فهذه التكلفة بسيطة جدًا مقارنة

بما يوفره. أما قرار اعتماد Magento تقليدي يعني إنفاق ربما \$50k\$ – 10k على تطوير الموقع واستضافة وتأمين قبل بيع أي منتج. بالطبع، مع نمو الأعمال الكبيرة جدًا، قد تصل رسوم SaaS الإجمالية (خاصة نسب العمولة من المبيعات) إلى أرقام أكبر مما لو كان المتجر يملك منصته الخاصة، وهذا ما قد يدفع بعض العلامات التجارية الكبرى مستقبلاً للانتقال إلى حل مخصص. إذن فالأمر يتعلق **بمقايضة بين تكلفة البداية وتكلفة الامتلاك الطويل**.

**باختصار في التكاليف:** SaaS يوفر **وفورات ملحوظة في التكلفة المبدئية والصيانة** لأنه يشارك البنية التحتية عبر عملاء كثر [14] ، ويناسب نماذج الأعمال التي تفضل الإنفاق التشغيلي الدوري. الأنظمة التقليدية تتطلب **استثمارًا أوليًا كبيرًا** لكنها قد تكون مجدية ماليًا على مدى طويل لبعض السيناريوهات مع قاعدة مستخدمين ضخمة ثابتة (حيث تكاليف الاشتراك قد تتجاوز كلفة حل داخلي). لذا ينبغي لكل شركة إجراء تحليل التكلفة والفائدة بناءً على وضعها الخاص.

## قابلية التوسع (Scalability)

**التوسع السريع والمرونة:** صُممت تطبيقات SaaS أصلاً لتكون **مرنة وذات قابلية توسع عالية** لتلبية احتياجات عملاء متعددين ومستخدمين كثر عبر الإنترنت. تستطيع البنية السحابية **التمدد أفقيًا بسهولة** – أي إضافة خوادم ووحدات معالجة إضافية تلقائيًا عند زيادة الحمل [55] [56] . هذا يعني أنه إذا تضاعف عدد مستخدمي الخدمة أو تضاعف حجم البيانات، يمكن للبنية أن تتكيف في الوقت الحقيقي تقريبًا عبر تشغيل مثيلات إضافية من التطبيق أو تخصيص موارد إضافية. **مرونة التوسع الآلية (Elastic Scalability)** هذه إحدى نقاط التفوق الكبرى لنموذج SaaS والبنى السحابية عامة [57] [58] . شركات تكنولوجيا كبرى مثل أمازون وجوجل استثمرت بقوة في هذا الجانب مما مكّن خدمات SaaS مثل Netflix وغيرها من خدمة ملايين المستخدمين حول العالم دون انقطاع.

**التوسع في الأنظمة التقليدية:** على العكس، **توسعة نظام تقليدي تتطلب تخطيطًا وترقية يدوية**. فإذا احتاجت شركة مضيفة لنظام داخلي أن تدعم مستخدمين إضافيين، قد تضطر لشراء خوادم ذات مواصفات أعلى أو إضافة خادم جديد ثم **نقل وتنصيب النظام عليه** . هذا قد يستغرق أسابيع أو أشهر في بعض المؤسسات (بين شراء العتاد، تكوينه، اختبار سلامة العمل عليه) [59] . كما أنه قرار مالي مهم (استثمار في عتاد) قد يُتخذ بناء على توقعات نمو ربما أو ربما لا تتحقق، مما يؤدي أحيانًا إلى **توفير مفرط** (Overprovisioning) – أي امتلاك سعة أكثر من اللازم معظم الوقت [59] . هذا الوضع غير فعال مقارنةً بسيناريو SaaS حيث تتم إضافة الموارد وتحريرها بشكل ديناميكي حسب الحمل الفعلي. لذلك نجد تطبيقات تقليدية كثيرة **محدودة بقدرات بنيتها** ؛ ربما تدعم 100 مستخدم جيدًا لكن لو صاروا 1000 يصبح الأداء سيئًا ما لم يتم مشروع ترقية شامل.

**التوسع العالمي:** في SaaS، بما أن الوصول عبر الإنترنت، يمكن **لأي عدد من المستخدمين عالميًا** استخدام الخدمة، ومزود الخدمة هو من يتعامل مع توزيع الحمل عبر المناطق إذا لزم الأمر. إذا توسعت شركة من سوق محلي إلى عالمي، فتكفي إضافة مراكز بيانات إضافية في مناطق جديدة (غالبًا يوفرها مزود السحابة تلقائيًا) دون تغيير في التطبيق نفسه. بينما نظام تقليدي كلما دخل سوق أو فرع جديد، يحتاج إما نشر نسخة جديدة في ذلك الموقع أو جعل المستخدمين يتصلون بالمقر الرئيسي عبر شبكة واسعة (WAN) والتي قد تعاني من تأخير. هذا يعطي **أفضلية واضحة لـSaaS في دعم التوسع الجغرافي** السريع للأعمال.

**القيود والبنية:** بطبيعة الحال، **قابلية التوسع ليست لا نهائية** . على مهندسي SaaS تصميم أنظمتهم بحيث لا توجد عنق زجاجة مركزية تحد من التوسع (مثل الاعتماد على قاعدة بيانات وحيدة عملاقة دون إمكانية التقسيم – Sharding). أفضل الممارسات تشمل **تجزئة البيانات أفقياً** (مثل توزيع مستأجرين مختلفين على قواعد بيانات مختلفة عند نموهم)، واستخدام **التخزين المخبئي (Caching)** لتخفيف الحمل عن الموارد الرئيسية [60] [61] ، وتوظيف خدمات سحابية قابلة للتوسع ذاتيًا (مثلاً استخدام خدمات مدارة مثل DynamoDB أو BigQuery التي تدير التوسع تلقائيًا). أيضاً الاستفادة من **الحوسبة بدون خادم (Serverless)** لبعض المكونات يمكن أن يعطي قدرة توسع شبه غير محدودة لسيناريوهات معينة بدون تعقيد إداري كبير.

في الأنظمة التقليدية، حتى لو أرادت شركة تطبيق مبدأ التوسع الأفقي، فقد تصطدم بتحديات مثل **تعارض الجلسات** في التطبيقات غير المصممة لذلك، أو الحاجة إلى **إعادة هندسة** لاستغلال عدة خوادم، مما يعني أن كثيرًا من التطبيقات التقليدية تظل محصورة ضمن خادم واحد قوي بدل توزيعها، وخاصة التطبيقات القديمة (Legacy).

**ملائمة التوسع حسب طبيعة العمل:** إن كان العمل المعني **قابلاً للنمو أو التقلص بشكل كبير** (مثلاً شركة تجارة إلكترونية تتوقع موسم ازدهار في نهاية العام ثم ركود نسبي)، فإن SaaS يمنح اطمئنانًا أنها تستطيع خدمة الزيادة المؤقتة دون مشاكل، ثم تعود الموارد لحالتها دون تكاليف دائمة. أما إذا كان حجم العمل **ثابتًا تقريبًا** ولا يتوقع تغييرات كبيرة في الحمل (مثال: تطبيق لإدارة الموظفين في شركة عدد موظفيها ثابت تقريبًا)، فميزة التوسع الديناميكي ليست حاسمة وقد يكون حل داخلي ثابت كافيًا.

**أمثلة رقمية:** منصة مثل **Slack** (خدمة مراسلات SaaS) صممت لتستوعب انضمام آلاف الفرق الجديدة وربما مئات الآلاف من المستخدمين يوميًا أثناء فترة نموها السريع، وتمكنت من ذلك عبر بنية خلفية موزعة تعتمد على خدمات AWS القابلة للتوسع. في المقابل، أنظمة مراسلة داخلية أقدم (مثل Microsoft Lync القديم) كان يتطلب توزيع المستخدمين يدويًا على عدة خوادم إذا زادوا عن حد معين، مع الكثير من الخطوات اليدوية. الفارق ظهر في تجربة المؤسسات التي انتقلت إلى Slack ووجدت أنها لم تعد تقلق حول سعة الخادم وعدد المستخدمين، مما يثبت **قيمة SaaS في التوسع المريح**.

## تجربة المستخدم (User Experience)

**إمكانية الوصول (Accessibility):** من منظور المستخدم النهائي، **القدرة على الوصول من أي مكان وفي أي وقت** هي إحدى أهم مزايا SaaS. المستخدم يحتاج فقط إلى متصفح أو تطبيق جوال واتصال إنترنت ليصل إلى تطبيقه وبياناته. لم يعد مقيدًا بالتواجد في المكتب أو على شبكة الشركة للوصول للنظام [7] . هذا غيّر شكل تجربة المستخدم تمامًا، حيث أصبح العمل **عن بعد والتعاون الموزع** ممكنًا بفضل SaaS. على سبيل المثال، فرق المحاسبة أو المبيعات تستطيع تحديث البيانات والحصول على التقارير لحظيًا من الميدان عبر أنظمة SaaS السحابية، بينما في النظام القديم كان يتوجب الانتظار حتى الرجوع إلى المكتب أو استخدام شبكات VPN معقدة للوصول للتطبيق الداخلي.

**التعاون والتشارك:** معظم تطبيقات SaaS مبنية مع أخذ **التعاون الفوري** في الحسبان. فغالبًا توفر **تحريرًا متزامنًا** (كما في مستندات Google) أو **رؤية لحظية لتعديلات الزملاء** (كما في منصات المحاسبة السحابية التي تعرض التغييرات مباشرة لكافة المستخدمين) [25] . هذا يخلق تجربة مستخدم أكثر **تفاعلية وترابطًا** ، ويزيل الحواجز بين المستخدمين. بالمقابل، الأنظمة التقليدية عادةً مصممة **لمستخدم واحد في الوقت الواحد** ضمن قاعدة البيانات؛ إذا احتاج عدة أشخاص العمل معًا، قد ينتج عن ذلك تعارضات أو يحتاجون إجراءات خاصة (كحجز سجلات أو تقسيم العمل يدويًا). قد تفتقر الحلول القديمة لإمكانية العمل التعاوني الحقيقي لعدم تصميمها الشبكي منذ البداية. لذا **تجربة الفريق على SaaS** أكثر سلاسة: الكل يرى نفس المعلومات المحدَّثة ويعمل على منصة واحدة دون الحاجة لإرسال ملفات أو دمج تغييرات يدويًا.

**واجهة المستخدم والتحديثات المستمرة:** بشكل عام، تطبيقات SaaS تميل إلى امتلاك **واجهات مستخدم عصرية وسهلة الاستخدام** أكثر من كثير من التطبيقات التقليدية. السبب أن SaaS كمنتج يخضع لمنافسة في السوق ويتوجب أن يجذب المستخدمين بواجهة جذابة وتجربة سلسة، كما أن الفريق بإمكانه **تحسين الواجهة باستمرار اعتمادًا على تحليلات الاستخدام** وإصدار تلك التحسينات لجميع المستخدمين فور جاهزيتها. مثلًا، وصف أحد المراجعين واجهة QuickBooks Online بأنها "أكثر سلاسة وحداثة وسهولة في الملاحة" مقارنةً بواجهة QuickBooks Desktop التي وصفها بالقدِمة والمعقدة [62] [63] . في الحقيقة، كثير من البرمجيات التقليدية خاصة القديمة تعاني من **واجهات قديمة** لأنها صُممت في وقت مختلف ولم تتح لها تحديثات جذرية خوفًا من كسر التوافق مع قواعد المستخدمين. على النقيض، SaaS يسمح بأخذ **ردود فعل المستخدمين بشكل مستمر** وإدخال تحسينات صغيرة متتابعة تجعل المنتج أكثر سهولة ومتعة في الاستخدام. أضف إلى ذلك أن SaaS غالبًا **يدعم الأجهزة المحمولة والمتصفحات المختلفة** تلقائيًا، في حين أن تطبيقًا مكتبيًا تقليديًا قد لا يتوفر له تطبيق جوال مكافئ مما يحد من تجربة المستخدم العصري الذي يحب التنقل بين أجهزته.

**الخPerformance وسرعة الاستجابة:** قد يعتقد البعض أن وجود التطبيق في السحابة قد يعني أداء أقل بسبب التأخر في الشبكة (Latency)، لكن في الواقع مع تحسينات البنى التحتية للإنترنت وتقنيات CDN والتخزين المخبئي، **يمكن لتطبيقات SaaS تقديم أداء ممتاز** . المزود السحابي يمكنه توجيه المستخدم لأقرب خادم، واستخدام موارد قوية لخدمة العديد من المستخدمين بكفاءة. بينما أداء النظام التقليدي يعتمد على موارد الخادم لدى العميل والتي قد تكون محدودة أو غير مُحَدَّثة دوريًا. كثير من الشركات الصغيرة مثلًا تشغل أنظمتها التقليدية على عتاد

متقادم مما يؤدي إلى تجربة بطيئة، في حين انتقالها إلى SaaS يقدم لها خوادم عالية الأداء دون استثمار مباشر فيها. بالتأكيد، تبقى **سرعة الإنترنت عاملاً** للمستخدم: فإذا كان الاتصال سيئًا، قد يعاني مع تطبيق SaaS، بينما تطبيق مكتبي محلي قد يعمل بسلاسة دون إنترنت. لهذا السبب، إحدى ميزات بعض الأنظمة التقليدية هي **العمل أوفلاين** ؛ فمثلاً QuickBooks Desktop أوفلاين تمامًا مما يعني أنه يمكن للمحاسب العمل أثناء رحلة طائرة أو انقطاع الشبكة [64] ، بينما QuickBooks Online يتطلب اتصالاً. بعض تطبيقات SaaS حلت هذا جزئيًا بتوفير وضع عمل غير متصل (Offline Mode) في تطبيقات الجوّال أو بآليات مزامنة، لكن بشكل عام **تعتمد SaaS على الإنترنت** مما قد يؤثر على التجربة في المناطق ذات الاتصالات الضعيفة أو في حالات انقطاع الشبكة.

**دعم المستخدم والتدريب:** غالبًا SaaS يأتي مع **تجربة استخدام موجهة** (Onboarding) مدمجة ترشد المستخدم الجديد خطوة بخطوة في التطبيق، لأن المزود لديه مصلحة بجعل المستخدم يفهم ويندمج بسرعة دون تدريب خارجي. رأينا هذا مثلاً في تجربة QuickBooks Online حيث كل خطوة يسبقها شرح وإعداد مبسط [65] ، بينما النسخة التقليدية كانت أقل إرشادًا وتتطلب اعتمادًا أكبر على خبرة المستخدم أو مواد تدريب منفصلة [66] . هذه الاختلافات الصغيرة تُحدِث فرقًا في **زمن تعلم النظام** وسرعة تحقيق الإنتاجية منه.

في المحصلة، **تميل تجربة المستخدم في SaaS إلى أن تكون أكثر إيجابية وحداثة** بفضل قابلية الوصول، والتعاون، والواجهات المحدثة باستمرار، بشرط توفر اتصال إنترنت جيد. أما **تجربة المستخدم في الأنظمة التقليدية** فقد تكون أفضل في حالات تتطلب عملًا دون اتصال أو تخصيصًا شديد التفرد، لكنها عمومًا تعاني من محدوديات البيئة المحلية (كعدم الوصول من خارج المكتب، أو واجهات قديمة، أو غياب الميزات التعاونية). لهذا نجد المستخدمين اليوم يتوقعون من البرمجيات نفس سهولة الاستخدام التي يلاقونها في تطبيقاتهم اليومية على الهاتف – وهذا توقع يحققه نموذج SaaS بوتيرة أسرع من البرامج التقليدية التي قد تبقى جامدة بعد التثبيت.

## الخلاصة

بعد استعراض كافة الجوانب من التحليل إلى التنفيذ، يتبين أن **نموذج SaaS والأنظمة التقليدية لكل منهما مزايا وتحديات مختلفة جوهريًا**. يمكن إجمال الفروق الرئيسية كالتالي:

- **جمع المتطلبات والتصميم**: يركز SaaS على بناء منتج مرن بميزات معيارية قابلة للتكوين لتلبية احتياجات عامة ومتطورة، بينما يتعمق النموذج التقليدي في تلبية متطلبات مخصصة عمقًا لمؤسسة محددة [2] .
- **المعمارية والبنية**: يقوم SaaS على بنى سحابية متعددة المستأجرين تسمح بتشارك الموارد والتوسع السهل، مقابل بنى تقليدية منعزلة لكل عميل تمنح تحكمًا كاملاً على حساب المرونة [5] .
- **إدارة المستخدمين والبيانات**: يقدم SaaS هيكلًا هرميًا للمستأجرين مع عزل صارم للبيانات عبر حلول برمجية، أما الحلول التقليدية فتستفيد من العزل الطبيعي لكل نشر منفصل إنما تفتقر لقدرات التعاون عبر المؤسسات.
- **النشر والصيانة**: يتفوق SaaS في سرعة الإعداد والتحديث المستمر بلا كلفة على العميل [19] ، بينما تستلزم الأنظمة التقليدية جهدًا تقنيًا من العميل في كل مرة تحديث أو توسعة مع سيطرة أكبر له على جدولة التغييرات.
- **الأمثلة الواقعية**: تثبت قصص نجاح مثل QuickBooks Online وShopify كيف يمكن لـSaaS إحداث تحول في الصناعة [24] [28] ، في حين تبقى بعض السيناريوهات لصالح الحلول التقليدية (كما في حالات الامتثال الشديد أو التخصيص العميق).
- **الأخطاء الشائعة**: يحتاج بناء SaaS إلى حذر وتخطيط لتجنب مطبات مثل عدم تضمين التعددية منذ البدء أو تفريق قاعدة المنتج لطلبات خاصة [3] [33] .
- **الأمان والامتثال**: يوفر SaaS أمنًا قويًا مبنيًا على خبرات المزود وشهاداته، إنما يتطلب ثقة العميل والتأكد من تلبية المتطلبات التنظيمية لكل قطاع [23] . الحلول التقليدية تمنح سيطرة مباشرة لكنها تلقي العبء كاملًا على المؤسسة في إدارة المخاطر.
- **التكلفة والتوسع وتجربة المستخدم**: يبرز SaaS بخفض كلفة الدخول والصيانة مع قابلية توسع فورية وتجربة استخدام حديثة ومتاحة عالميًا [55] [7] ، على عكس الحلول التقليدية ذات التكلفة الأولية العالية والتوسع المحدود محليًا وتجربة المستخدم المقيدة بالمكان والزمان.

في النهاية، **اختيار بين SaaS والنظام التقليدي يجب أن يتم بناءً على احتياجات العمل وظروفه الخاصة**. قد يكون **SaaS مثاليًا** للشركات التي تبحث عن سرعة الانتشار، والكلفة المرنة، والتحديث الدائم، والتعاون السهل، لا سيما في بيئات تعتمد تقنيات الويب الحديثة (مثل فرق التطوير التي تستخدم Node.js وReactg وغيرها). بالمقابل، قد تفضل بعض المؤسسات **الحلول التقليدية** عندما تكون أولوياتها السيطرة الكاملة على البيانات، أو وجود متطلبات تخصيص وامتثال لا يمكن الوفاء بها إلا داخل أسوارها الرقمية. كما يوجد خيار **الحلول الهجينة** والجمع بين النموذجين (مثل استخدام SaaS لبعض الوظائف الثانوية وحل داخلي للمهام الأساسية الحرجة).

من الواضح أن البرمجيات كخدمة أصبحت **اتجاهًا رائجًا ومسيطرًا على سوق البرمجيات** نظرًا لما توفره من قيمة مضافة وسرعة مواكبة للتطور [32] . ومع استمرار نضج التقنيات السحابية، ستتضاءل الكثير من فجوات القلق حول الأمان والامتثال. بالمقابل، سيظل للأنظمة التقليدية مكان في سيناريوهات محددة. على صناع القرار الموازنة بين هذه الفروق الجوهرية لاتخاذ القرار الصحيح الذي يخدم إستراتيجية أعمالهم وتطلعاتهم المستقبلية في التحول الرقمي.

**المصادر:**


1. Kenny & Company – *"As Simple as SaaS? Not so FaaSt"* – مناقشة تبني منهجية المشاريع التقليدية في مبادرات SaaS [2] [20] .
2. Milvus AI Reference – *"How does SaaS differ from traditional software?"* – شرح الفروق في التسليم والتحديث والتكلفة بين النموذجين [1] [19] .
3. InfoQ – ملخص محاضرة *"Mistakes People Make Building SaaS"* – أفضل الممارسات وتجنب الأخطاء الشائعة في بناء تطبيقات SaaS [3] [33] .
4. Forbytes Tech Blog – *"Single Tenant vs Multi Tenant SaaS Architecture"* – شرح نماذج تعدد المستأجرين ومزايا وعيوب كل منها [13] [17] .
5. Ardoq Blog – *"SaaS vs On-Premise: Informed Decisions"* – مقارنة شاملة عبر التكلفة والصيانة والمرونة والأمان بين SaaS والأنظمة المحلية [67] [55] .
6. Zapier – *"QuickBooks Online vs. Desktop (2025)"* – مراجعة مقارنة من منظور المستخدم لتجربة QuickBooks السحابي مقابل المكتبي [24] [7] .
7. Human Element – *"SaaS vs. On-Prem for eCommerce"* – دليل لاختيار منصات التجارة الإلكترونية مع مقارنة بين Shopify (SaaS) Magentog (تقليدي) [30] [31] .
8. Medium – *"Scaling Node.js for Multi-Tenant Architectures"* – مناقشة تحديات تصميم تطبيقات متعددة المستأجرين في Node.js مع استراتيجيات قاعدة البيانات [9] [6] .
9. SentinelOne – *"Cloud vs On-premise Security: 6 Differences"* – مقارنة جوانب أمنية مثل التحكم في البيانات والتخصيص والموثوقية بين السحاب والبيئات المحلية [40] [41] .
10. مجموعة ريناد المجد (مدونة عربية) – "ما هي الحوسبة السحابية؟" – مقال يذكر أن الحلول السحابية قد تكون أكثر أمانًا من الأنظمة التقليدية لدى الكثير من المؤسسات الصغيرة [23] .


---

?How does SaaS differ from traditional software 46 19 1
https://milvus.io/ai-quick-reference/how-does-saas-differ-from-traditional-software

As Simple as SaaS? Not so FaaSt 20 2
/https://michaelskenny.com/points-of-view/as-simple-as-saas-not-so-faast

QCon London: Mistakes People Make Building SaaS Software - InfoQ 38 37 36 35 34 33 32 5 3
/https://www.infoq.com/news/2025/04/qcon-saas-software-mistakes

Scaling Node.js for Robust Multi-Tenant Architectures | by 61 60 12 11 10 9 8 6 4
Arunangshu Das | Medium
https://article.arunangshudas.com/scaling-node-js-for-robust-multi-tenant-architectures-feecb05de7b7?gi=095691ec9357

QuickBooks Online vs. Desktop: Which is best? [2025] 66 65 64 63 62 27 26 25 24 21 7
/https://zapier.com/blog/quickbooks-online-vs-desktop

Single Tenant vs Multi Tenant SaaS Architecture - Forbytes 42 18 17 16 15 14 13
/https://forbytes.com/blog/single-tenant-vs-multi-tenant

SaaS vs On-Premise: Making Informed Software 67 59 58 57 56 55 54 48 47 45 44 43 22
Decisions | Ardoq
https://www.ardoq.com/blog/saas-vs-on-premise

23 ما هي الحوسبة السحابية؟ دليلك الشامل لاستغلال قوة السحابة في 2025
-https://www.rmg-sa.com/%D9%85%D8%A7-%D9%87%D9%8A
-D8%A7%D9%84%D8%AD%D9%88%D8%B3%D8%A8%D8%A9%
/D8%A7%D9%84%D8%B3%D8%AD%D8%A7%D8%A8%D9%8A%D8%A9%

SaaS vs. On-Prem for eCommerce – Your Guide to Choosing a Platform | Human 53 52 31 30 29 28
Element
/https://www.human-element.com/saas-vs-on-prem-for-ecommerce-your-guide-to-choosing-a-platform

Cloud vs On-premise Security: 6 Critical Differences | SentinelOne 51 50 49 41 40 39
/https://www.sentinelone.com/cybersecurity-101/cloud-security/cloud-vs-on-premise-security