LLMOps for Lean Product Teams

Authors:

Aoife Hughes

Samuel Adebayo, PhD

ISx4 • AI research - 2025

ISx4
AI & ANALYTICS
Insight, Strategy, Solutions, Services & Support

info@ISx4.com - http://www.ISx4.com

Executive Summary

In this whitepaper, we define LLMOps as a new operational discipline distinct from traditional MLOps, explain why it has emerged alongside foundation models, and show how lean product teams can turn prototypes into reliable, compliant, and cost-aware products; we then compare external (API-based), internal (self-hosted/fine-tuned), and hybrid approaches – clarifying benefits, trade-offs, and when to use each – followed by a practical visual framework that maps the moving parts of LLMOps (data & retrieval, prompt/version management, evaluation, CI/CD, observability, cost & latency controls, governance, and incident response); we then ground the concepts with sector snapshots (fintech assistants, customer-support copilots, internal productivity tools) that balance rapid experimentation with enterprise-grade reliability through versioned prompts, offline/online eval, canary & A/B testing, feedback loops, and privacy controls; and we conclude with an evolution path from external-first to hybrid/in-house as volume, customization, or regulatory needs grow – without derailing roadmaps. The takeaway for AI engineers and enterprise tech leads: LLMOps is the playbook for moving from demos to durable products. If you're ready to operationalize LLMs with clear guardrails and measurable outcomes, isx4 can partner end-to-end – from strategy and architecture to build, launch, and ongoing optimization.

Table of Contents

Title

Introduction

The rise of large language models (LLMs) like GPT-3, GPT-4, and others, has revolutionised natural language processing, enabling powerful applications from chatbots and code assistants to enterprise search and document analysis[1]. However, taking these models from an impressive prototype to a reliable, scalable, and secure product is far from straightforward[1]. This challenge has given birth to LLMOps (Large Language Model Operations) – a methodology for managing the entire lifecycle of LLMs in production environments[2]. In essence, LLMOps narrows the focus of traditional MLOps (Machine Learning Operations), to address the unique challenges of deploying and maintaining LLMs (e.g. OpenAI's GPT series, Google's PaLM/Gemini, Anthropic's Claude) at scale[2]. It encompasses specialised methods and processes to accelerate model creation, fine-tuning, deployment, monitoring, and continuous improvement over an LLM's lifespan[3]. Notably, LLMOps has rapidly emerged as a distinct field since early 2023, mirroring the surge in generative AI adoption in industry[2].

For lean product teams – small, agile groups aiming to incorporate AI capabilities without enormous infrastructure – LLMOps represents a new paradigm. It brings DevOps/MLOps best practices into the era of LLMs, ensuring that even a nimble team can develop, deploy, and iterate on LLM-powered features efficiently. In short, while a lone developer can quickly prototype an LLM-based feature using an API, LLMOps is what enables a team to productise that prototype with confidence. It imposes structure (for prompt design, data handling, model versioning), provides tools for monitoring and feedback, and keeps the entire process reproducible and adaptable.

The LLMOps landscape is evolving fast, comprised of tools and techniques such as:

- **LLM-as-a-Service APIs:** Providers like OpenAI, Azure, or Google offer hosted LLMs via API calls, allowing teams to leverage powerful models without managing infrastructure.[3]
- **Custom LLM Stacks:** Open-source models (e.g. Llama 2, GPT-J) and libraries that teams can fine-tune and deploy on their own infrastructure for proprietary solutions.[3]

[1] What is LLMOps and how does it work? - Weights & Biases
[2] What is LLMOps? Lifecycle, benefits and challenges
[3] What is LLMOps? Key Components & Differences to MLOPs

- **Prompt Engineering Tools:** Frameworks for crafting and refining prompts, enabling in-context learning as an alternative to fine-tuning.[3] This includes prompt version control and chain-of-thought orchestration utilities.
- **Vector Databases:** Specialised databases for semantic embeddings that allow relevant context retrieval (retrieval-augmented generation) to ground LLM responses in up-to-date internal data.[3]
- **LLM Orchestration Frameworks:** Tools like LangGraph, Akka, or LlamaIndex that help string together multiple LLM calls and integrations (e.g. with search or calculators) into coherent LLM pipelines or agent-like behaviors.[3]
- **Monitoring & Feedback Systems:** New monitoring solutions to track LLM-specific metrics (token usage, latency, quality of outputs) and capture human feedback or detect issues like hallucinations in production.[3]

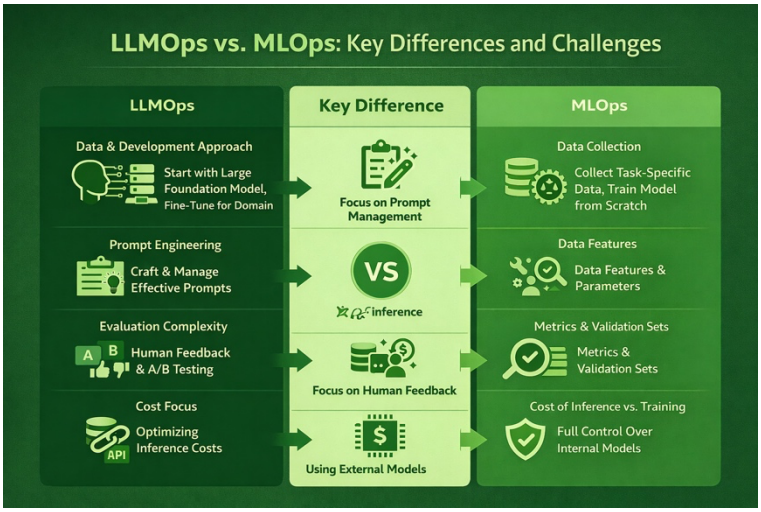
All these components fall under LLMOps. In the following sections, we'll explore how LLMOps differs from classical MLOps and examine strategies for lean teams to utilise LLMs – whether using external model services, building models in-house, or a hybrid of both. The goal is to show why LLMOps is not just a buzzword, but a necessary evolution in AI product development, and how even a lean team can harness it to stay competitive.

LLMOps vs. MLOps: Key Differences and Challenges

LLMOps builds upon the foundation of MLOps, but introduces new considerations that reflect how LLM-based AI products are built differently from traditional ML products[1]. Many core DevOps/MLOps principles still apply (for example, version control, CI/CD pipelines, environment separation[4]), but the development and deployment of LLMs pose unique challenges. Below are some key differences that make LLMOps a distinct model:

- **Data and Development Approach:** Traditional MLOps often involves training models from scratch on task-specific data or fine-tuning smaller pre-trained models. In contrast, LLMOps heavily relies on transfer learning. Lean teams typically start with a large foundation model and fine-tune it on domain-specific data, rather than collecting massive raw datasets to train a new model from the ground up[4]. In many cases, teams even skip fine-tuning initially and use the model as-is via prompts. This means LLMOps must handle model selection (from model hubs or providers, internal or external) and manage iterations that might involve gradually moving from third-party models to custom fine-tuned models[4].

- **Prompt Engineering as a First-Class Citizen:** A unique aspect of LLMOps is the importance of prompt design and management. Since LLMs can perform tasks given the right instructions or examples in the prompt, teams spend significant effort on crafting effective prompts or prompt templates. Prompt versioning and experimentation is a core part of LLMOps, something that has no real equivalent in traditional MLOps[3]. Entire pipelines of prompts and tools (using frameworks like LangChain) may be built to achieve a desired outcome, treating the prompt itself as a piece of "software" to be improved. MLOps, on the other hand, rarely deals with anything like prompt orchestration – it focuses on data features and model parameters. LLMOps must track prompt changes, test prompt variations, and ensure consistency of LLM behavior across prompt updates[1].



[4] LLMOps workflows on Databricks | Databricks on AWS

- **Evaluation Complexity:** Evaluating a classic ML model usually means comparing predictions to ground truth labels using well-defined metrics (accuracy, F1, etc.). With generative LLMs, there is often no single correct answer, making evaluation much trickier[1]. Automated metrics like BLEU or ROUGE exist for certain tasks (e.g. translation or summarisation), but they are imperfect proxies for quality[3]. As a result, LLMOps pipelines rely heavily on A/B testing, user feedback, and human evaluation to assess model performance in real-world scenarios[1]. For example, teams might deploy two prompt variants or model versions and compare user ratings of their outputs. Continuous human-in-the-loop feedback (such as collecting user corrections or preferences) becomes essential to gauge and improve an LLM's usefulness[3]. This is a departure from traditional MLOps, which can often optimise a metric on a validation dataset and be confident in model performance.

- **Cost Focus (Inference vs. Training):** In classical machine learning projects, the major costs tend to come from data labelling and model training compute, whereas running the model (inference) is comparatively cheap per prediction. With LLMs, this is flipped: large models incur *substantial computational cost every time they generate output*, especially if using a paid API or if the prompts/outputs are lengthy[1]. LLMOps must therefore emphasise **cost monitoring and optimisation at inference time** – e.g. tracking token usage, caching frequent results, truncating unnecessary lengthy outputs – to keep the solution economically viable[1]. Lean teams also need to be mindful of the costs of experimentation: calling an API like GPT-4 repeatedly during development racks up expenses, which is a new concern not seen in small-scale ML model training. MLOps, by contrast, was more concerned with the one-time training cost and scaling out infrastructure for model deployment; LLMOps has to manage ongoing usage-based costs and possibly use techniques like model distillation or prompt optimisation to reduce inference load[3].

- **Latency and Performance Constraints:** Large language models are, by nature, computationally heavy. Serving an LLM (especially a big one with billions of parameters) can introduce noticeable latency for end users – e.g. waiting several seconds for a response – which can hurt user experience. This makes **latency optimisation a key part of LLMOps**[1]. Techniques include using faster/optimised model architectures, prompt truncation or simplification, and deploying models on GPUs or specialised hardware for faster inference. In MLOps, latency is also a concern, but typically model inference was a smaller fraction of the overall pipeline time; with LLMs, generating text *is* the main workload and often significantly slower than retrieving a prediction from a typical ML model[1]. Moreover, LLMOps might involve **streaming responses** (sending tokens as they are generated) to improve perceived latency, something not relevant to classic ML. For lean teams, careful budgeting of latency vs. model size is important – sometimes opting for a slightly smaller model that responds faster can be a wiser product decision.

- **Operational Constraints with External Models:** Another difference is that many LLM-powered applications depend on **third-party model providers**. When using an external LLM service (e.g. via an API), teams lack the low-level access to model internals. This makes some MLOps practices (like traditional model versioning and artifact tracking) harder or require new approaches. If the provider updates the model behind the API, you might suddenly get different outputs or performance. Rolling back to a previous version isn't always possible if the service doesn't support it[3]. **Model observability** also changes: instead of your own model logs, you rely on whatever telemetry the API gives. LLMOps thus must include **API governance** (keeping track of model versions offered by a provider, evaluating new versions, and having a strategy if an update regresses performance)[3]. We will discuss this more in the external vs internal section, but it's a noteworthy shift – you might treat the *prompt* or the *chain* as the versioned artifact in your pipeline, rather than the model itself, when the model is out of your control[4].

In summary, LLMOps can be viewed as **"MLOps 2.0" tailored for large language models**, carrying over core principles of collaboration, automation, and continuous improvement, while adding new layers for prompt management, human feedback integration, cost/latency optimisation, and dealing with the quirks of enormous pre-trained models[3] [5]. The field is still evolving rapidly as best practices coalesce. Next, we'll look at how lean teams can approach LLMOps through different strategies – using external services, building in-house, or hybrid combinations – along with their trade-offs.

External LLMOps: Leveraging Third-Party LLM Services

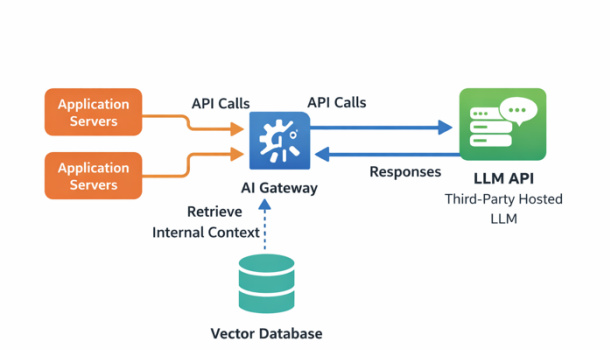


Figure : Example production architecture for a retrieval-augmented application using a third-party LLM API as the language model. The application servers interact with an external hosted LLM via API calls (through an AI gateway), optionally enhanced with a vector database for retrieving internal context to include in prompts.

One attractive approach for lean teams is to use **external LLM providers** – essentially outsourcing the heavy lifting of the language model to services like OpenAI, Microsoft Azure OpenAI Service, Anthropic, Cohere, etc. In this **External LLMOps** model, the team integrates an API into their product: you send inputs (prompts) to a third-party model and get back outputs, without ever training or hosting the model yourself. This has **clear**

[5] LLMOps Demystified: How to Make LLMs Work in the Real World | by Sahin Ahmed, Data Scientist | Medium

benefits for a small or fast-moving team: it drastically cuts down on upfront work and infrastructure. You can tap into a state-of-the-art LLM within hours, enabling rapid prototyping and development. It also offloads the maintenance of model upgrades and scaling to the provider. *In other words, using a ready-made LLM can be more cost-effective in the short term and gets you to market faster*[6].

However, relying on external LLMs comes with **trade-offs** that LLMOps must manage:

- **Customisation Limitations:** A third-party model might not perfectly fit your domain or use case out-of-the-box. Fine-tuning may not be available (or may be limited to what the provider allows), so you often have to use prompt engineering to coerce the model's behavior. This means if you need highly domain-specific knowledge or a distinct style, an external model could fall short. The model is essentially a one-size-fits-many solution. Some providers do offer fine-tuning on their models, but it can be expensive and you still don't get full control. In summary, you sacrifice a degree of customisation in exchange for convenience[6].

- **Data Privacy and Compliance:** Sending data to an external API raises concerns about sensitive information. Lean teams in regulated industries (finance, healthcare, etc.) must consider whether user data or proprietary data can be safely sent to a third-party cloud. Even if the provider claims to secure your data, there may be legal or customer perception issues. **If your business deals with sensitive information, using an external LLM service requires caution** – you'd need assurances (and perhaps contracts) about data not being stored or used to train the provider's models[35]. Some companies solve this by only sending non-sensitive prompts externally or by anonymising/encrypting parts of queries, but those add complexity.

- **Operational Dependency:** When the core intelligence of your product lives outside your own stack, you are inherently dependent on that provider. If the external service has an outage or degradation, your application is directly impacted. Likewise, any changes the provider makes (model version updates or API changes) can affect your app's behavior. Notably, you **lose some control over versioning** – if the API switches to a new underlying model version, you might not be able to roll back to the old one if problems arise[3]. This makes testing and monitoring even more important: teams should constantly evaluate the outputs as the external model may drift or change without your input. Robust LLMOps for external models might include *contract tests* for the API (to catch changes) and keeping a close eye on provider announcements.

- **Cost Management:** While you avoid the cost of training a model, **inference cost can be significant** with external services. Most providers charge per token or per request. For a popular feature, these costs can accumulate rapidly. LLMOps for external

[6] Should You Build or Buy Your LLM? – TextMine

models needs to include budgeting and monitoring of API usage, perhaps setting up alerts if costs exceed thresholds. In some cases, it may even become cost-effective to bring the model in-house if usage is very high – a classic build-vs-buy calculation. Lean teams should be mindful of this tipping point. As Chip Huyen observed, the cost of long or complex prompts manifests at inference time[1], so prompt optimisation (keeping prompts as concise as possible while achieving the task) is not just a performance concern but a cost-saving strategy.

- Despite these challenges, external LLM services remain a **powerful option for lean teams** to get started quickly. Many successful LLM-driven products began by leveraging an API like GPT-3/GPT-4 to test the waters and deliver immediate value. **LLMOps in this context focuses on the surrounding glue**: how to feed the right prompts, how to post-process model outputs, how to monitor quality, and how to ensure reliability around an essentially black-box model. A real-world example is a startup building a customer support chatbot: by using a third-party LLM API, they can *focus on crafting great prompt templates and integrating with their knowledge base*, rather than spending months training a model. The key is to put in place the operational guardrails – monitor the responses for accuracy and safety, log conversations for analysis, handle exceptions – so that the service feels robust to end-users even though the "brain" is outsourced.

- **Latency and Integration Overheads:** Calling an external API introduces network latency and overhead that wouldn't exist if the model were local. Each request has to travel to the provider and back, which can add significant delay, especially if you need to send large prompts or use streaming. This can impact user experience. Additionally, integration means handling **API credentials, rate limits, and costs**. You must securely manage API keys and perhaps build a mechanism to avoid hitting rate limits or gracefully handle them. Databricks, for example, notes that when using external LLM APIs, you incur added complexity around authentication and potential latency, which must be accounted for in your architecture[4]. Good practice is to implement caching of results when possible and to design your system to degrade gracefully if the external API is slow or unavailable.

Internal LLMOps: Building and Managing In-House Models

At the other end of the spectrum, lean teams may opt for Internal LLMOps, where the large language model is owned and operated by the team itself (or the organisation). This typically means using open-source LLMs or model weights that you can host, and customising them to your needs – through fine-tuning, prompt tuning, or other adaptation techniques – and then deploying them on your own cloud or on-premise infrastructure. The internal approach offers maximum control and potentially long-term cost benefits, but it also demands greater expertise and resources upfront.

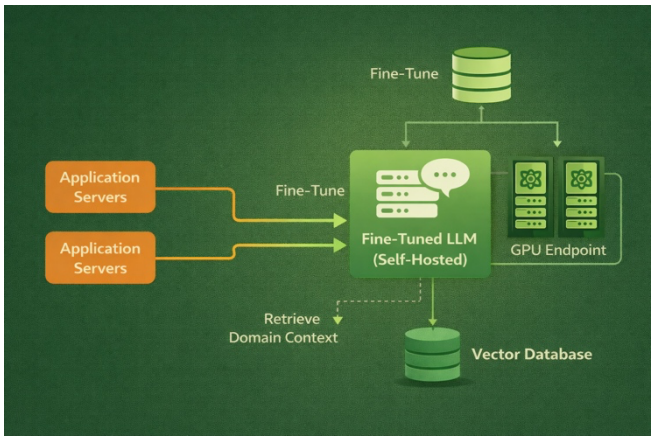


Figure: Example production architecture for a RAG application using a self-hosted, fine-tuned open-source LLM. Here the team fine-tunes an open model on domain data, then deploys it on their own infrastructure (with GPU endpoints for serving), while using a vector store for retrieval. This represents an internal LLMOps approach with full model control.

Why go internal? For some teams, the motivations include:

- **Customisation and Domain Expertise:** By building your own LLM (or fine-tuning an open one), you can tailor the model's knowledge and style exactly to your domain. You are not constrained by a third-party model's limitations. For instance, you can train on proprietary data (clinical notes, legal documents, internal codebase, etc.) to imbue the model with knowledge that general models lack. You can also enforce style guidelines or company-specific terminology. This level of customisation can give a competitive edge or better user satisfaction, as the model's outputs align closely with what you need[6]. Internal models can also be iteratively improved: if you find a weakness, you can gather more data and fine-tune again, an option often not available with closed APIs.

- **Privacy and Compliance:** Keeping the model in-house means **your data stays in-house** during both training and inference. This is crucial if the content of prompts or the knowledge the model uses is sensitive. Industries dealing with confidential information (user personal data, financial records, healthcare data) may simply *not be able* to send that data to an external service due to regulations or policies. An internal LLM allows these teams to still leverage LLM capabilities without compromising on privacy[6]. Many companies also have compliance requirements (GDPR, HIPAA, etc.) that are easier to meet when the data never leaves their controlled environment. By building the model themselves, they can also ensure that **the model's outputs do not inadvertently leak training data**, since they know exactly what went into it.

- **Avoiding Vendor Lock-in:** With an in-house model, you are not tied to the roadmap or pricing of a vendor. Organisations sometimes prefer this route to have **long-term independence**. If you've fine-tuned an open-source model, you could even port it to different cloud providers or scale it up/down freely without worrying about API rate limits or sudden price hikes. Your team's know-how becomes a core asset. This can be strategically important for enterprises who see AI models as part of their IP. It also means you can inspect and audit the model's behavior more directly, which might be useful for troubleshooting or for responsibility (knowing why the model said something, which is still challenging but at least you have access to weights and training data internally).

That said, the **internal LLMOps path is challenging**, especially for a lean team:

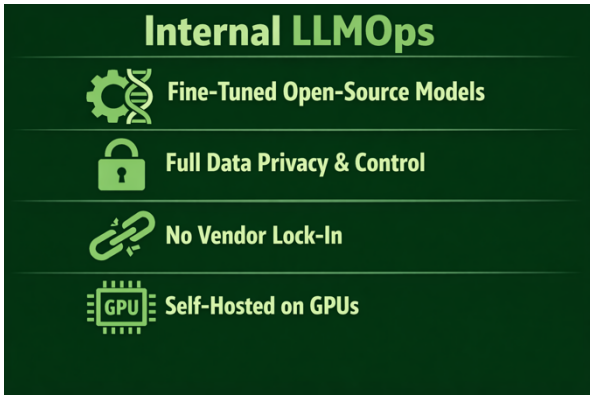
- **Significant Expertise and Resources Required:** Building or fine-tuning LLMs is not trivial. It requires knowledge of machine learning and deep learning frameworks, and often specialised skills in working with large models (distributed training, GPU memory management, etc.). You'll need ML engineers or AI scientists who can navigate this. Moreover, **the computational resources needed are substantial** – large GPU servers or cloud GPU instances, possibly tens of thousands of dollars' worth of compute to fine-tune a multi-billion parameter model, depending on the size of data and epochs[6]. Even optimising a model via techniques like quantisation or low-rank adaptation (LoRA) requires careful work[6]. For a lean team, these requirements can strain the budget and timelines. As a rule of thumb, teams should only embark on this if they have (or can acquire) the expertise in-house and if the use case is critical enough to justify it.

- **Longer Time to Market:** Compared to plugging into an API, building your own LLM **takes much more time**. There is model selection (choosing an appropriate open-source base model), data gathering and cleaning for fine-tuning, the fine-tuning process itself (which might involve trial and error to get right), and setting up inference servers for deployment. All this could mean weeks or months before you have a production-ready model, whereas an external API could be integrated in days[6]. Lean teams should weigh this delay against the benefit of customisation. In fast-moving markets, a slow launch could be a big disadvantage. One practical approach is to use an external model initially (to launch sooner) and work on an internal model in parallel – essentially a hybrid strategy we'll discuss later.

- **Maintenance and Operations Overhead:** Once you own the model, you also own the **ongoing maintenance**. This includes monitoring its performance, handling model drift as new data comes in, and updating the model when needed to improve or to include new knowledge[6]. Large models may need periodic retraining or fine-tuning as your data distribution changes or if new better architectures become available and you want to migrate. You'll also need to set up proper **model versioning and**

governance internally: tracking which model version is deployed, what data and hyperparameters were used to train it, and ensuring reproducibility. All of these practices are part of LLMOps for internal models, and they require disciplined effort. Lean teams must be careful to automate as much as possible (e.g., using MLOps platforms or pipelines) to reduce manual workload. Additionally, considerations like bias and fairness of the model fall on your shoulders – you must evaluate and mitigate these during development[6], whereas with an external model you somewhat outsource that responsibility (though you still need to monitor outputs for inappropriate content either way).

- **Infrastructure Costs:** Serving a large model can be expensive. If your product needs to handle many users or requests, you might have to scale out GPU servers or use specialised hardware (like AWS Inferentia or Azure ND series instances, etc.). The cost of running these 24/7 can exceed the cost of using an API, depending on scale. There's a breakeven point: at very high volume, self-hosting might be cheaper, but at low to moderate volume, paying per use might be cheaper. Lean teams should perform cost analysis – sometimes hosting a smaller open model that's "good enough" could save money over hitting an expensive API for every request. Techniques like model quantisation can help reduce serving costs (by using less memory and compute) if you go internal[6]. Nonetheless, budgeting for infrastructure (and having DevOps capability to manage uptime, scaling, deployments of the model servers) is an important part of internal LLMOps.



In practice, **internal LLMOps for lean teams often means starting with smaller-scale models or highly targeted fine-tuning**. For example, a company might take an open-source 7B-13B parameter model and fine-tune it on their proprietary data to achieve a model that, while not as generally powerful as GPT-4, performs exceptionally well on the company's niche tasks. Such a model can be hosted with a few GPU instances and serve the needs without incurring per-request fees. The team would then build all the usual trappings around it: an API or service interface, monitoring for model responses, logging and analytics to gather user feedback, etc., similar to any ML service.

Internal LLMOps gives **full control to experiment** – if the model is hallucinating, you can try techniques like adding retrieval augmentation, or fine-tune on counter-examples; if it's too slow, you can attempt distillation to a smaller model. You essentially have the model as another piece of your codebase. The downside is you are responsible for it end-

to-end. Lean teams that succeed here typically leverage existing open-source tooling (for example, using Hugging Face Transformers libraries, or platforms like MLflow to track experiments, etc.) to not reinvent the wheel in their operations[3].

To summarise, **going internal is a strategic choice**: it can yield differentiated capabilities and control in the long run, but requires careful investment. A lean team must ensure they have a clear business case (e.g., regulatory necessity or a feature that absolutely requires a custom model) and then proceed methodically with LLMOps best practices to implement it. Otherwise, the effort might derail. Many teams eventually adopt a hybrid approach – combining internal and external – which we discuss next.

Hybrid Approaches: Combining External and Internal LLMOps

Not every organisation will strictly choose all-in external or all-in internal. In fact, a **hybrid LLMOps approach** is increasingly common, especially for lean teams trying to maximise benefit while mitigating risks. Hybrid can refer to a couple of different patterns:

1. **Architecture Hybrid:** Using external LLM services for some functions while using internal models or data sources for others, within the same product.
2. **Phased Hybrid (Build-vs-Buy Timeline):** Starting with external LLMs to get off the ground, then gradually **bringing the model in-house** (or deploying a fine-tuned open model) once the product matures or cost/privacy factors demand it.

Hybrid in Architecture (Mixing API and In-House): In a single application, it's possible to use an external API for what it's best at, and complement it with internal components. A prime example is *retrieval-augmented generation (RAG)* systems. A team might use a powerful external LLM to handle the natural language generation, but **feed it data from an internal vector database** containing the company's proprietary knowledge. In this setup, the heavy language understanding is done by the external model, but it's grounded on internal data that never leaves your environment (only relevant snippets or embeddings are sent to the model) – achieving a balance of performance and privacy. For instance, a customer support bot could call an external GPT-4 API but always prepend retrieved relevant content (e.g., an internal FAQ or policy document excerpt) to the prompt. The user's query and the retrieved context go out, but sensitive databases are never exposed externally. This *hybrid LLMOps* requires orchestrating both an internal search/index component and external model calls together. Tools like LangChain facilitate such orchestration, and monitoring must cover both parts (e.g., ensuring the retrieval component finds good info, and the external model uses it correctly).

Another architectural hybrid example: an AI assistant might use an **external LLM for general conversation or reasoning**, but for certain tasks that involve confidential logic, it switches to an **internal model**. Say a financial planning app uses an external LLM for chit-chat and basic questions, but when it comes to advising on a user's sensitive investment data, it queries an internally fine-tuned model that knows about finance and

is deployed in the company's VPC. From the user's perspective it's one assistant, but behind the scenes the system routes requests either externally or internally based on content. This can be orchestrated by an application layer that detects the query type. The benefit is you use the external model's strength for broad knowledge, and your internal model for proprietary insights or compliance. The complexity lies in maintaining two systems and deciding which to use when.

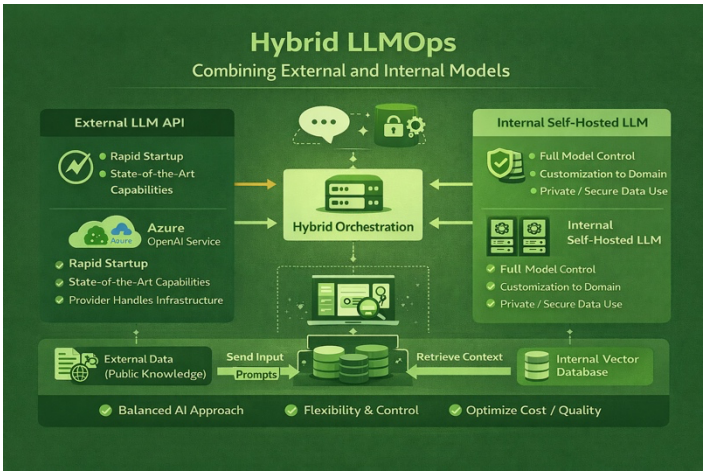
Hybrid in Phases (Evolution over Time): Many lean teams take a pragmatic trajectory: **start external, then internal later if needed**. Early on, when exploring a product idea, the priority is to validate the concept quickly. External APIs are perfect for that – no ML team needed, just integration and prompt engineering. As the product gains users, the team collects data on how the model is used, what the shortcomings are, and also how costs scale. They might discover that it's getting expensive, or that the model often fails on some domain-specific queries. At that point, they consider training or fine-tuning a custom model to address those gaps or to reduce dependency. Over time, they might transition more of the workload to an internal model. For example, they might fine-tune an open-source LLM on all their accumulated conversation logs to better handle their users' queries. Initially, that fine-tuned model could be used as a fallback or for certain requests, but eventually, if it becomes good enough, it might replace the external API entirely, or run alongside it to split the load. **LLMOps supports this phased approach** by enabling continual evaluation and comparison. A best practice is to run the new internal model in shadow mode first – i.e., it generates answers in the background for some queries while the external model's answer is still shown to users, and the team compares performance. This way, they can ensure the in-house model meets a quality bar before fully switching over. Tools for experiment tracking and model registry become important here to manage multiple models in play[1].

Ultimately, a hybrid strategy aims to **get the best of both worlds**: the *agility and power of external LLMs* and the *control and custom tailoring of internal models*. Many industry experts advocate that for complex real-world applications, a combination is often most effective[5] [7]. For example, fine-tuning a base model on domain data can give you a strong foundation in your niche, while retrieval augmentation keeps the model's knowledge up-to-date without retraining[1]. In fact, research and practice are showing that combining these techniques yields better outcomes than relying on either alone: *fine-tuning provides the domain fluency and adherence to requirements, whereas retrieval (or other prompt-time tooling) provides factual grounding and flexibility*[1]. One source notes that for many production use cases, *"hybrid approaches offer the best of both: factual accuracy with domain-specific tuning"*[1].

[7] LLMOps in Production: 287 More Case Studies of What Actually Works

For lean teams considering hybrid approaches, here are a few **tips**:

- Start with clear criteria for what stays internal vs goes external (e.g., "anything involving customer personal data will use our internal model" or "the external API will be used for tasks above a certain complexity threshold until we can handle them internally"). Defining these boundaries helps architecture and compliance.
- Design your system to be modular, so you can swap out or add models easily. If using a middleware or gateway (like an API orchestrator), it can route to different backends (one for external, one for internal) without the client or UI needing to change. This also allows A/B testing different approaches seamlessly.
- Monitor both cost and quality metrics continuously. Perhaps the external API is more expensive but has higher quality responses – you might choose to send only queries of certain types to it to optimise cost-performance tradeoff. Or if your internal model starts outperforming, you shift more traffic to it. The decision can be dynamic.
- Keep security in mind on both fronts: secure the integration with external providers (encrypt data in transit, minimal scopes for API keys) and also secure your internal model environment (ensure the model server is behind authentication, data used to train is properly governed).



- Have a fallback plan. If your internal model crashes or lags, maybe you can temporarily fail over to the external API to maintain service (even if at higher cost). Conversely, if the external service has issues, your internal model might handle some requests. A hybrid setup can provide redundancy if orchestrated well.

In summary, **Hybrid LLMOps** is a pragmatic approach for many lean teams: it lets you leverage cutting-edge AI quickly, but also invest in proprietary capabilities where it counts. By combining external and internal strategies, teams can innovate rapidly without betting the farm on one approach. This often leads to more robust and cost-efficient systems.

LLMOps in Action: Use Case Highlights

To ground these concepts, let's briefly look at a few scenarios across different sectors, illustrating how lean product teams might apply LLMOps:

- **Customer Support Chatbots (External-first to Hybrid):** Imagine a small SaaS company that wants to add an AI chatbot to its website to answer user questions. Initially, the team integrates an external LLM API and uses prompt engineering with a few example Q&As. This covers general queries well. As the bot gets used, they find it sometimes struggles with very product-specific questions. The team then builds a retrieval component: they index their product documentation in a vector database. Now their chatbot (hybrid system) will fetch relevant doc snippets and include them in the prompt to the external LLM, improving accuracy. Over time, as usage grows, they fine-tune an open-source LLM on past chat logs and deploy it internally to reduce API calls. The result is a **cost-effective chatbot** that uses an internal model for known questions and falls back to the external API for particularly hard or open-ended questions. Throughout, LLMOps practices (monitoring chat transcripts, measuring resolution rates, updating the prompt templates) ensure the bot's performance keeps improving.

- **Healthcare Data Assistant (Internal-focused):** A small healthtech startup aims to help doctors summarise patient visit notes and retrieve insights from medical literature. Due to strict privacy needs, they choose an internal LLMOps approach. They fine-tune a pre-trained clinical language model on de-identified patient notes to learn the style and terminology. They also connect the system to an internal database of medical research articles via an embedding-based search, so the model can pull in facts for evidence-based answers. Here, **internal LLMOps** involves careful data handling (ensuring no PHI leaks in training data), compliance checks, and model validation with clinicians. The lean team uses MLOps tools for experiment tracking to compare different fine-tuning configurations and uses human feedback (doctor reviews of summaries) to further refine the model. The outcome is an AI assistant that the team can **safely deploy in hospitals**, as it doesn't send data externally and is tuned to the healthcare domain. The trade-off was a slower development period and the need for ML expertise on the team, but it meets the domain's demands.

- **Enterprise Knowledge Base Copilot (Hybrid cloud):** Consider an enterprise software company with a lean innovation team tasked with improving employee productivity with AI. They want a chatbot that can answer any question about the company's internal operations (HR policies, IT support procedures, etc.). They decide to use Azure's OpenAI service (external) for the language capabilities but keep all company documents in an internal Azure storage and use an Azure cognitive search (vector search) to provide context to the model. Because everything runs in their Azure cloud tenant, it feels internal, but the model is essentially external (OpenAI's). This hybrid solution is quick to set up using cloud-managed services. The LLMOps focus is on curating the document index and crafting robust prompts that include retrieved data and instructions to keep answers factual. They also set up monitoring to flag any responses that might reveal sensitive info, adding an extra layer of

governance. The result is a **company-internal chatbot** that employees can query for information, leveraging a powerful model without the company exposing its data or developing a model from scratch. If later the company decides to fully internalise, they could replace the API with an open-source model hosted on Azure, but initially this hybrid cloud approach delivers value fast.

These examples show that **LLMOps is versatile** – it's about combining the right mix of tools and practices to reliably deliver LLM-driven functionality in your product. Whether your team uses external models, internal models, or both, LLMOps thinking ensures you address the **full lifecycle**: from data preparation and prompt design to deployment, monitoring, and feedback integration. Even for lean teams, there are now numerous platforms and open-source tools to help manage this complexity (from model training suites to prompt management and monitoring dashboards). The key is to adopt an experimental, iterative mindset: start with something small, instrument it well, learn from real-world use, and continuously refine the model or prompts. This is exactly what DevOps brought to software – continuous improvement – now applied to the AI model powering your product.

Conclusion and Call to Action

LLMOps has quickly become an essential discipline for any team looking to turn large language models into lasting product success. LLMOps is enabling teams to harness LLMs effectively, reliably, and responsibly. This is a response to real challenges and needs that arose when LLMs moved from research labs into customer-facing applications. Since the breakthrough of ChatGPT and similar models, we've seen an explosion of prototype integrations; now, the industry focus is on robustly productising these AI capabilities. As one industry leader put it, *"2023 was canonically the year of prototypes, and 2024 is shaping up to be the year of production."*[8] In other words, now is the time to get serious about LLMOps.

For lean product teams, this means opportunity. You don't need a giant staff or a huge budget to start applying LLMOps principles. Begin by assessing your current or upcoming AI features: How will you manage prompts and inputs? Do you have a way to measure output quality and get user feedback? Are you tracking the usage and costs of any third-party AI APIs? If you fine-tune a model, can you reproduce that model's build and roll it back if needed? These are the kinds of questions LLMOps urges you to answer.

[8] LLMOps Insights: Evolving GenAI Stack

Embrace an LLMOps mindset in your next AI project. Start with small steps – for example, implement version control for your prompts or set up a simple logging dashboard to review model outputs and user interactions; there are so many tools and framework to do these now. If you're using an external API, put in place a monitoring script to watch for latency or errors and to log the model's responses for analysis. If you're training or fine-tuning a model internally, use a tool (even a spreadsheet or simple database to start) to record your experiments, hyperparameters, and results. These practices will pay off quickly as you iterate. Additionally, consider leveraging community resources and tools: many open-source libraries and platforms (from experiment trackers and model hubs to prompt management tools and evaluation harnesses) are available to accelerate your LLMOps journey[3]. Don't reinvent the wheel if you don't have to – plug these into your workflow so you can focus on what makes your product unique.

Finally, cultivate a culture of continuous learning and improvement. LLM technology is evolving at lightning speed. New models, techniques (like better fine-tuning methods, RLHF improvements, etc.), and best practices are emerging constantly. Staying engaged with the LLMOps community – reading case studies, sharing lessons, experimenting with new tools – will help your lean team punch above its weight. In the end, successful LLMOps is about marrying the creativity of what an LLM can do with the rigor of engineering discipline. With the right approach, even a small team can build and scale incredible AI-powered products.

Now is the time to start. Whether you deploy an AI assistant for your users or an internal tool for your employees, bring LLMOps principles into the project from day one. By doing so, you'll save yourself pain down the road and deliver a better experience to your users. Lean teams have the advantage of agility – combine that with LLMOps know-how, and you can iterate to greatness faster than larger competitors. We encourage you to take the next step: identify a promising use case, apply the concepts discussed in this paper, and begin your LLMOps journey. The companies that master LLMOps will be the ones that consistently deliver AI features that delight users and stand the test of time. Will yours be one of them?