O'REILLY®

Early Release

RAW & UNEDITED

Building AI Agent Platforms

Tools and Tactics to Scale AI Applications Across the Enterprise

Ben O'Mahony & Fabian Nonnenmacher

Building AI Agent Platforms

Tools and Tactics to Scale AI Applications Across the Enterprise

Ben O’Mahony and Fabian Nonnenmacher

Building AI Agent Platforms

by Ben O’Mahony and Fabian Nonnenmacher

Published by O’Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

Acquisitions Editor: Louise Corrigan

Development Editor: Melissa Potter

Production Editor: Christopher Faucher

Cover Designer: Karen Montgomery

Interior Designer: David Futato

April 2027: First Edition

Revision History for the Early Release

- 2026-01-15: First Release

See https://oreilly.com/catalog/errata.csp?isbn=9798341666382 for release details.

979-8-341-66634-4

[LSI]

Brief Table of Contents (*Not Yet Final*)

Chapter 1. AI Applications and Their Lifecycle

"The question of whether Machines Can Think (…) is about as relevant as the question of whether Submarines Can Swim."

—Edsger Dijkstra

The release of ChatGPT in November 2022 has fundamentally changed the software industry. Not only was this a major catalyst for using Generative Artificial Intelligence (GenAI) applications and AI-based coding assistants to develop software, but it also has dramatically accelerated the development of *"AI applications"*.

While developing software with machine learning models is not new, ChatGPT's widespread availability was a revolutionary step. It enabled the creation of applications that were unimaginable for many developers just a few years ago and significantly lowered the barrier to entry, making AI application development more accessible than ever before.

This triggered a gold rush atmosphere, with startups and major corporations all scrambling to build new applications. The pace of innovation is staggering, with a new Large Language Model (LLM), framework, library, or development practice being announced on a near-weekly basis.

While it may feel like you are in uncharted territory, many of the best practices from traditional software engineering are still relevant. Too many people are needlessly reinventing the wheel.

An effective AI applications platform cuts through the hype, enabling your organisation to not only build AI applications quickly, but also in high quality. To understand the essential capabilities of such a platform, first let's dive into what AI applications are and identify the specific challenges during their development.

This chapter begins by defining AI applications, use cases in which they are used, and typical architecture patterns they follow. Next, it compares the application's lifecycle and that of a traditional software application. It explains how the discipline of AI Engineering can be seen as the continuation of the DevOps principles. Finally, it highlights key similarities and differences to software and outlines how AI Engineering specific challenges are best addressed, contrasting them with common anti-patterns.

What is an AI application?

At its core, an AI application is any software application that provides features built on top of artificial intelligence models. This means that building AI applications is a cross-functional activity, requiring developers, data scientists, and domain experts to work together.

The term *AI* (Artificial Intelligence) includes a wide range of technologies, from simple rule-based systems to complex neural networks. While academically incorrect, with recent innovations and the dominance of Generative AI in the machine learning field, the term often is used as a synonym for Generative AI. This book will also follow this convention. Whenever AI applications are mentioned in this book this refers to applications built with generative models.

Generative models are a specific type of Machine Learning (ML) model that can generate new content, whether it be text, images, audio, or video. AI applications create value by leveraging those generative capabilities in the implementation of their features. These applications differ significantly from those that integrate with more traditional machine learning models, such as fraud detection algorithms or recommendation engines. This is mainly due to the variety of potential inputs and outputs. The preceding examples would output a True or False and maybe a confidence score, whereas a generative model could output any text you could imagine. This leads to different challenges and thus requires specific strategies to deal with them.

It's also important to clarify that we are not referring to applications that were simply built with the help of AI-powered coding assistants. However, these coding assistants are, in fact, AI applications themselves [1].

An AI application fundamentally leverages Generative AI capabilities to provide user-facing features and can be anything from a simple script or chatbot, to image generators, to complex assistant systems. Importantly, the process that leverages Generative AI doesn't necessarily need to be exposed to the end user for an application to be an AI Application (Though maybe don't market it as so unless the AI is relatively obvious).

Foundational Models

In the context of AI applications as discussed in this book, a more widely used term for generative models is often *'foundation models'*. It's important to note that while many large generative models are foundation models, the latter term is broader, encompassing any large, pre-trained model designed for adaptation to a wide range of tasks.

Different from traditional machine learning models that are trained and optimised for a single, specific task, foundation models are general-purpose models. This means they can be applied to a wide variety of contexts without needing to be retrained, which makes them the *'fundamental'* building block for AI applications.

While Large Language Models (LLMs) are the most well-known example, foundation models are not limited to just text. They can be multimodal, meaning they can both understand and generate content across different data types, such as text, images, audio, and video.

Large Language Models

As LLMs are the most prominent example of foundation models, this section takes a deeper look at them. They focus exclusively on text, so they are easier to grasp than the more complex multimodal models. At the same time, other multimodal models work similarly.

It's fair to admit that easier is relative here. Given that AI models are at the forefront of modern machine learning research, this book won't provide an in-depth explanation of how LLMs work.

However, as they form the foundational building blocks of AI applications, it is important to get a basic understanding of their core mechanics. This will help get an intuitive understanding of their characteristics, which define the specific challenges that need to be addressed during the development of AI applications.

At its core, an LLM is a probabilistic text completion engine. It takes an input text (prompt) and predicts the completion of the provided text by using a probabilistic approach.

WARNING

Most AI Application Programming Interfaces (APIs) are billed by the token, so it is vitally important to understand how they work!

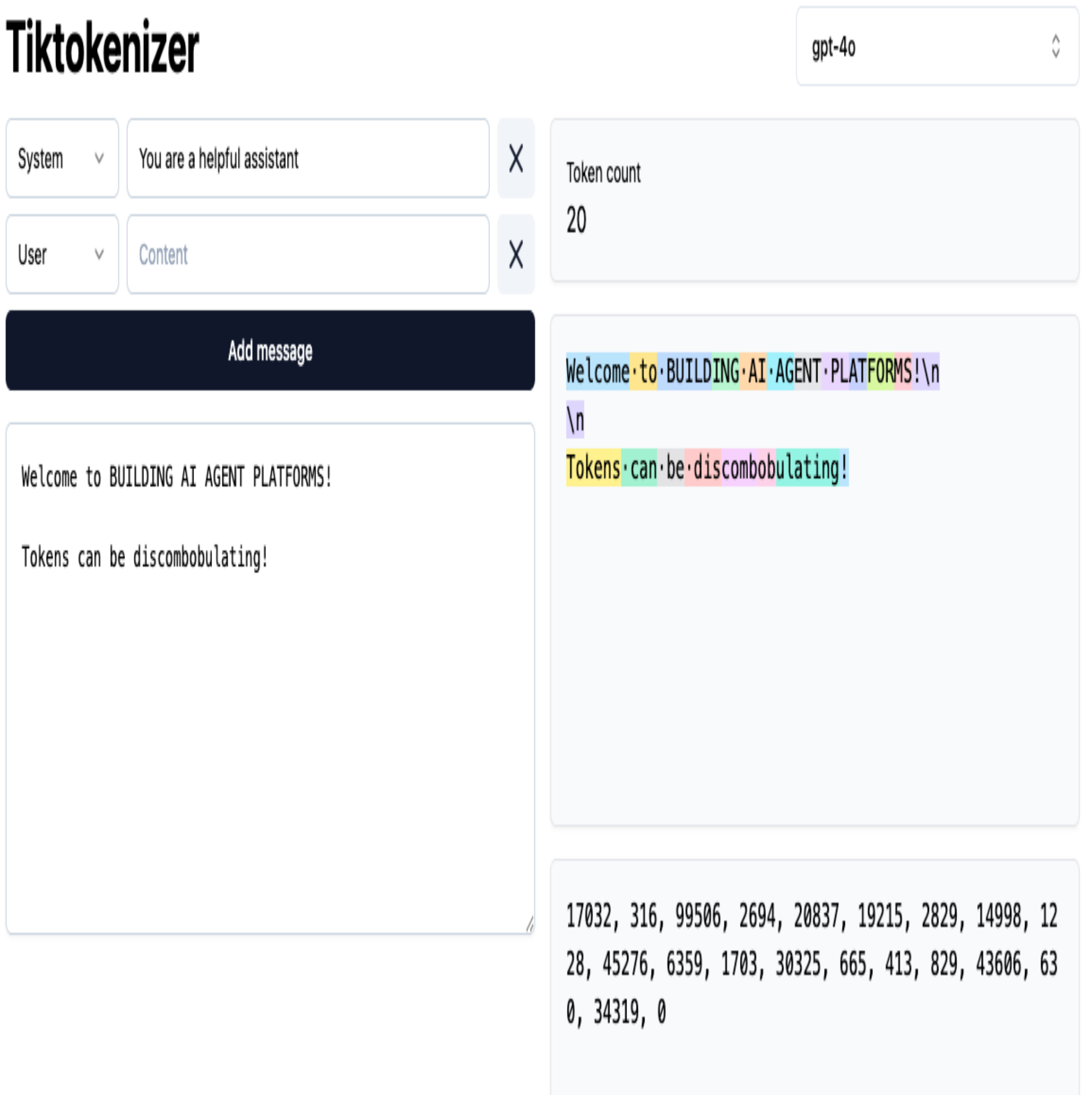
It is probably enough to know that words ≠ tokens and in fact the number of tokens is often much higher.

For a deep dive into the topic and see how this works in code, watch Andrej Karpathy's YouTube video Let's build the GPT Tokenizer.

Instead of using words, LLMs operate on *tokens*.

The original way of encoding text was to use a fixed vocabulary of characters. However, this ended up being too inefficient, needing a full prediction for every single character. This led to encoding the text into a fixed vocabulary of tokens, as visualised in Figure 1-1. This basic unit of an LLM can be any length string, but they are usually a character, word, or part of a word. Those tokens help break down words into more meaningful parts, e.g. cooking into *cook* and *ing*, with both parts representing a certain meaning.

The encoding is to pair these string tokens with a unique identifier; a commonly used algorithm for this is Byte Pair Encoding (BPE). In BPE, an encoding model is trained on a large corpus of text and learns how to pair tokens together most efficiently. This allows the LLM to predict the next token in the sequence more efficiently and also allows for the model to be more compact.



```
Welcome to BUILDING AI AGENT PLATFORMS!

Tokens can be discombobulating!
```

```
Welcome·to·BUILDING·AI·AGENT·PLATFORMS!\n
\n
Tokens·can·be·discombobulating!
```

```
17032, 316, 99506, 2694, 20837, 19215, 2829, 14998, 1228, 45276, 6359, 1703, 30325, 665, 413, 829, 43606, 630, 34319, 0
```

Figure 1-1. Example of Tokenization using Tiktokenizer

Today, most LLMs internally follow the transformer architecture introduced in "Attention Is All You Need" (Vaswani et al., 2017). This architecture introduces an attention mechanism that allows an LLM to not only consider the last word, but basically the full input for the predicted output.

LLMs implement different decoding strategies to select the next token. Many of the commonly applied strategies are not purely deterministic. Often there is a random element that helps decide on the continuation of the output. This makes the process non-deterministic. Meaning the model does not always produce the same output for a given prompt.

While intuitively this might sound unnecessarily complex, it is actually essential to make LLMs powerful and adaptive. For example, you can retry if you are not satisfied with the given output. Probably something everybody has done already when using ChatGPT's user interface or similar applications.

More importantly is that this makes LLMs *'creative'*. This means that an LLM generates output that sounds novel or surprising which is very valuable.

For an intuitive understanding of this concept, consider the input *The meal was absolutely…*. A high probable continuation is something like *… delicious*, but by not picking the most probable continuation, the model might answer with *…a symphony of flavors where the basil played the cello*.

This creativity comes with a downside. It might lead to more *inconsistencies* (a model gives contradicting answers) and *hallucination* (a model gives answers not based on facts). Therefore, the development of AI applications is a lot about managing and controlling these negative side effects. And with this, it's essential for an AI agent platform to provide the right tools to approach those challenges.

While LLMs *just* being a completion engine might sound simple, text generation actually is an extremely powerful capability. This capability already enables many other use cases, such as translation, summarization, or generating code. As those tasks often can be reframed into a completion challenge. For example, the prompt *In Python, a hello world program looks like…* might be completed with the output *print('Hello World*)', effectively being a code generation.

Use Cases of AI Applications

While the pace of model development continues to scale, along with their capabilities, the number of use cases for AI applications is growing but lagging behind. Many enterprises have gone all in on AI without the deep focus on use cases required to drive adoption. Therefore, shifting the emphasis to identifying and prioritizing adoption-driving use cases is a vital strategic imperative.

Coding

One of the breakthrough use cases for GenAI is coding assistance. Code has many elements that have made this an excellent use case for AI. Firstly, it is a very expensive, but also valuable process; this means that any improvements will have a significant impact on the organisation’s bottom line. Secondly, there are many ways of validating poor quality AI output, ranging from basic functional checks (like whether the code runs or compiles) to a plethora of advanced code quality tools and metrics. Finally, there is a motivated human in the loop to ensure the quality of the output and to provide feedback on the AI’s output, either directly or indirectly.

NOTE

These are the key criteria that drive the adoption of GenAI: valuable, easy to validate quality, and a motivated human in the loop.

The best example of this use case is AI powered auto-completion.

This has emerged as an incredibly popular tool for developers, allowing them to write code faster. Whenever the generation of a code snippet is not satisfactory, the developer can simply keep typing, and the AI will continue to generate a new snippet. As soon as an acceptable snippet is generated, the developer can accept it with a simple *tab* or '<C-y>.’[2] This acceptance of the generated code snippet is an excellent signal that the AI is producing a high quality output.

Moving beyond auto-completion, many startups built AI-powered Integrated Development Environments (IDEs). Owning the entire IDE application allows much more customization and control over building plugins to existing IDEs. In practice this resulted in many forks of Visual Studio Code with several incredibly valuable companies being created.

Still looking for more efficiency the next category created was Autonomous Coding Agents (Vibe Coding).[3] This was the idea that you don't even need to see the code, you just need to describe the task and the AI will do the rest. There are some very real advantages to this approach, and it certainly will continue to improve. As a rule of thumb it is amazing to build proof of concepts (POCs) and spike out ideas this way. However, these Agents go completely against the idea of small, focused changes to the code and so a huge amount of work is required to ensure quality for production. Ironically it seems like good quality and documentation practices that have been talked about for decades are also the things that predict the success of these Agents.[4] Maybe it will finally be possible to devote engineering time to fixing these issues? Either way this is an area to watch closely in the future.

Later chapters provide much more detail about the specifics of these Agents and how they can be used to build and maintain the platform.

Image and video production

Content Generation was one of the first use cases for image / video generative AI applications. While they have often struggled keeping context consistently over long videos, this is something that is being solved by the frontier labs. Much of the content now consumed online has, at least in part, been generated by AI.

Content moderation is the ying to the yang of content generation. Being able to detect and remove inappropriate content is a key challenge for content generation. Using Generative AI to moderate content is an interesting new tool.

Clip creation is slightly more simple than full content generation. With the rise of TikTok and other short form video platforms, short form content is on the rise. Re-using existing content is a key strategy for content creators. Enabling this quickly and easily is clearly a lucrative market and quality can be quickly understood based on the taste of the content creator or just the views that the content receives.

Writing

Writing is a humongous field with a wide range of niches and sub-fields. Of course LLMs are applicable here given their focus on text and language. However, because of the size of the field there is no such thing as a global writing quality. Each area has its own unique challenges and requirements and different styles of content for those areas will have their own preferences and eccentricities.

One of the early use cases for AI applications was summarization. Condensing large amounts of text into a shorter summary is very useful and can save a lot of time and effort. As before there is a motivated human in the loop who can fact-check to ensure the summary is accurate. There has been a huge proliferation of summarization tools in the last few years, and the field is still growing.

Technical writing is closely related to coding, where there are many hard rules that can be applied to understand quality. This can range from the use of a specific style guide, to the use of a specific vocabulary, to the fact that the content must be structured in a certain way. Once you layer in legislation and regulations, you have a very complex problem space but one that you can definitely get feedback on.[5]

Creative writing is a much more loosely defined field. Still there are narrative structures that can be applied to this. There are definitely some use cases here, but it is probably not applicable to the likely readership of this book.

Transcription and Translation

Again given the *language* element of LLMs, it is very easy to see how they can be used to transcribe and translate audio and video. This was already an area of research for machine learning and so there are lots of excellent datasets and tools available to help with this.

Almost every video conferencing app build in some kind of meeting summarization tool. In fact, today in 2026 within technology companies, it is rare to see someone taking notes in a meeting anymore. This has been a huge productivity boost for many people.

Translation is another area where LLMs have had a large impact. The complex nature of translation has made *perfect* translations a challenge, if they even exist. However, LLMs have made machine translation a lot easier and more accessible to many more people. Still, users would be better served by leveraging specific translation models and ideally incorporating validation from a native speaker to ensure quality.

Education

GenAI is excellent for learning using Socratic methods. You can ask questions and get answers from the model. The possibility, to ask follow up questions and dive in deeper, allows for a hugely customizable learning experience. There is also zero risk of humiliation for asking *stupid* questions. This has led to the development of AI Tutors in many different fields.

There is a lot of research into the use of GenAI for education. Without an expert to guide the learning process, it is very easy to go down rabbit holes and get lost. However, combined with many other learning methods this can be a very powerful tool.

Information aggregation

Chat with your Documents was one of the first use cases for GenAI. The ability to upload several documents and ask questions about them is a very powerful capability. It was also relatively easy to build simple Proof oc Concepts (POCs) of this. Nowadays, this has entered the enterprise search space with all the problems that come with that field.

Deep Research seems to be a promising use case for GenAI. Computers can read and write significantly faster than humans. This allows the aggregation of huge amounts of unstructured data to answer research questions. This capability has proven incredibly useful and valuable in practice.

Data organization

Structuring unstructured data is an area where AI applications shine. There are too many examples to mention including an agent to extract data from internal contracts for the legal team that would dramatically reduce manual effort.

Using an AI Application to really *understand* the business domain and to create an Ontology of the data is a very powerful capability. Using this Ontology as a semantic layer for the organisations data can unlock a much more nuanced analysis. Development of this capability is still relatively early, but it is definitely something that merits further exploration.

Workflow automation

This is where many enterprises have focused their efforts, hoping to automate their workflows and reap the efficiency gains. As with Robotic Process Automation (RPA), the worst thing to automate is something that should never have existed in the first place. Enterprises should begin with mapping processes and value streams, and then build the workflows that will drive the value. Going directly to full auto is often an error but engaging with the business to co-build tools and workflows is a great way to get started.

Automating Customer Support was an early use case for GenAI. Despite the fact that it is a very expensive and valuable process, it was not easy to validate poor quality AI output. Customer support is a very complex process and has often already had a lot of optimisation and automation attention, so there isn’t a lot of low-hanging fruit. AI Agents can actually be very effective with the long tail of customer support but many of the flagship success stories have since been debunked.

There will be many more of these use cases in the future, and with the benefit of experience using AI they have a much higher likelihood of success.

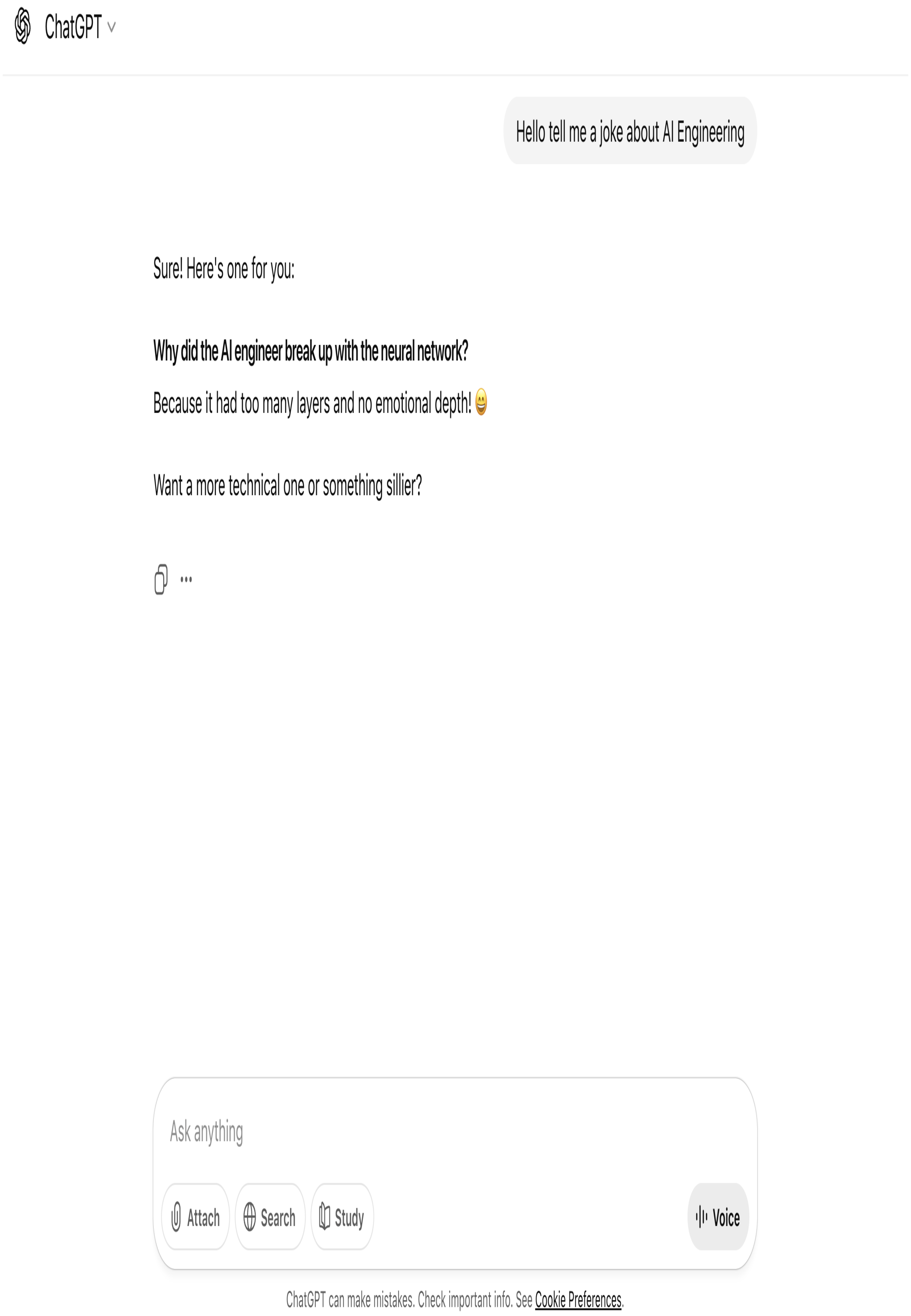
Common AI Application Patterns

After having discussed foundational models, the basic building block of AI applications, let’s see how those are integrated into applications through common patterns.

LLM Chat Interfaces

LLMs are text completion engines, you provide them with a snippet of text, and they continue it. For humans, that is not a very intuitive interaction. A human user would have to frame a question like, *What is your favorite food?* into the beginning of a text such as *My favorite food is…*.

The initial breakthrough in user accessibility, which also helped ChatGPT to gain its early popularity, was to abstract this behind a familiar chat interface (Figure 1-2). This additional abstraction layer allows for a more natural, conversational experience by converting a user’s prompt into a format the LLM can complete.



Hello tell me a joke about AI Engineering

Sure! Here's one for you:

Why did the AI engineer break up with the neural network?

Because it had too many layers and no emotional depth! 😄

Want a more technical one or something sillier?

ChatGPT can make mistakes. Check important info. See Cookie Preferences.

Figure 1-2. ChatGPT User Interface

In its most simple form, that would look like the following:

```
System Message: You are a helpful assistant
User: What do you recommend I have to eat?
Assistant: ...
```

By completing this text input, the LLM actually provides the assistant's answer. As of today, in many cases, this does not have to be done manually anymore. Most leading LLM providers (e.g., OpenAI, Anthropic, and Google) offer a dedicated chat endpoint which in the background formats the individual messages to a structured chat history.

Rather than simply concatenating text, those APIs use special role-based tokens to signalise the origin of each piece of text - whether it's from the system, the user, or the assistant. During the *instruction fine-tuning* [6] process, the LLM has learned about these special tokens, and thus the likelihood for the LLM to adhere to the expected format is increased.

Instead of just passing the latest user message to the LLM, it's common practice to pass the full conversation history to the endpoint. This technique gives the LLM *knowledge* of the conversation, and it can *remember* the ongoing conversation threads. Naturally, there is a limit. There is a maximum amount of text (tokens) an LLM can process, called the context window limit. As soon as the conversation history exceeds this limit, either old messages need to be dropped or the conversation needs to be summarised.

As there are many LLM providers or open source solutions offering those chat interfaces out of the box, there typically is little value in building a dedicated AI application that only wraps the plain LLM interactions with such a chat interface. However, chat interfaces are still an important pattern, as they have become a common interaction pattern also used in more complex AI applications.

Retrieval Augmented Generation applications

LLMs are limited to the knowledge that they have seen during training time. This *general knowledge* often does not contain the knowledge necessary for a specific domain or problem space in which the AI application is operating. Consider you want to build an application that can support interacting with an internal system, then typically the documentation of this system is domain-specific knowledge which probably was not included in the LLM's training data.

Retrieval Augmented Generation (RAG) is a technique that leverages fetching specific knowledge at runtime and provides it as additional input to the LLM. By providing this additional knowledge, RAG is a common technique to reduce hallucination and overcome training cut-offs. It was first introduced by Meta in the 2020 paper "Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks".

As the name indicates, RAG works in three steps (Figure 1-3). The first *retrieval* step takes the user input and retrieves additional information. Next, the initial user input is *augmented* with the retrieved information, which then is fed into an LLM that *generates* the final response considering the additional provided information.

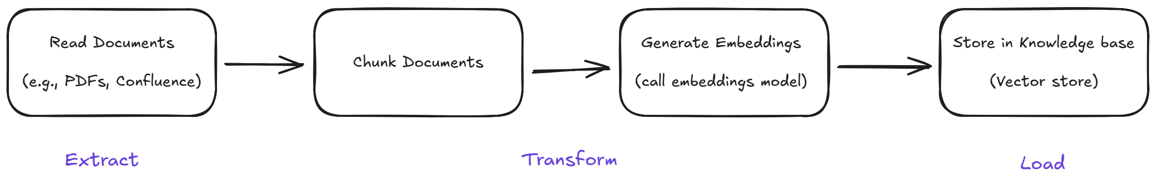


Figure 1-3. Retrieval Augmented Generation (RAG) architecture pattern

The information source within RAG is often referenced as the *knowledge base*. In the most simple form, the *knowledge base* is just a database. However, as RAG can be combined with all possible information retrieval techniques, RAG is an extremely powerful pattern that exists in many variations.

In practice, it's most commonly used with vector databases, which Chapter 5 covers in more depth. This is because there is a nice synergy between how foundation models and vector databases work. Both transform text into tokens (by using so-called *embedding models*), which then can be stored as vectors.

Storing these vectors in a database allows querying for similar vectors. Different from a keyword-based search, this is not based on exact string matches. Instead, due to the characteristics of the tokens, this allows finding entries that are *semantically similar*. For example, querying for *car* might also return entries that contain *automobile* or *vehicle*.

As of today, there are many off-the-shelf (or low code) solutions that build RAG applications internally with highly abstracted interfaces. Solutions such as OpenAI's Knowledge or AWS Bedrock's Knowledge bases allow uploading the additional information, e.g. as PDF, and abstract away the complexity of storing and retrieving the data.

Those off-the-shelf solutions are proven valuable for simple Q&A chatbots. However, once the domain data moves beyond simple tabular data, and more complex understanding and data modeling is required, those solutions quickly reach their limits. Additionally, RAG is a very fundamental pattern that is used in variations in more complex AI applications. Those variations often come by their own names such as Agentic RAG, multimodal RAG, adaptive RAG, or speculative RAG to just name a few.

AI Agents

As of today, AI agents are currently viewed as the most promising frontier in AI application development. While some AI evangelists may overstate their immediate potential to replace human labor completely, they are indeed a powerful technology. They allow automating complex, open-ended tasks that before were hard to automate efficiently.

The foundational concept of an AI agent, as established by Stuart Russell and Peter Norvig in Artificial Intelligence: A Modern Approach (Prentice Hall, 1995) is defined as anything that perceives its environment and acts upon it through actuators.

From a software engineering perspective, this can be reformulated to mean an AI agent is an application that can take decisions autonomously. Important to highlight, its decisions aren't based on a simple defined condition. Instead, the agent interprets complex, often unstructured information, typically with the help of an LLM. Broken down, this means an agent can define its own control flow at runtime, analyzing its inputs (*environment*) and autonomously deciding which specific functions or external tools (*actuators*) to invoke to achieve a certain goal.

A very simple agent designed to process incoming emails might look like this:

Example 1-1. A simple email processor AI Agent

```
def process(email):
    prompt = f"""
    Consider this email:
    {email}
    should it be ‘marked as spam‘, ‘marked as important‘
      or left alone?"
    """
    response = send_to_llm(prompt)

    if "marked as spam" in response.lower():
        mark_as_spam(email)
    elif "marked as important" in response.lower():
        mark_as_important(email)
```

Admittedly, this basic example would not strictly require an LLM.

It could easily be implemented using a traditional machine learning classification model or maybe even based on simple keyword filtering. However, it nicely illustrates the internal mechanics of the agent's autonomy.

AI agents become especially powerful when the task involves complex control flows. This includes, for example, tasks that require multiple sequential actions to be completed or iterative processes (loops) that require analysing the intermediate results before deciding on the next step. Using LLMs for making these decisions becomes particularly valuable when it is technically (or economically) not feasible to implement the necessary dynamic control flow with traditional, fixed software approaches.

The specific actions an agent can call are widely referred to as *tools*. These represent (parameterised) software functions that an agent can call autonomously. For instance, in Example 1-1, the tools available to the agent were `mark_as_spam` and `mark_as_important`. Given the importance of tools in modern AI applications, most LLM providers have incorporated native tool definition and calling directly into their APIs. Additionally, this usually involves representing them with specific tokens or structured output during post-training, which in practice has eliminated the need for error-prone string-matching (as in the simple example).

Moreover, the need to write custom logic for connecting the LLM with software functions (tools) has significantly diminished. There are now many publicly available software libraries and frameworks that manage this *glue logic*. They offer user-friendly abstractions for defining tools and automating the tool calling process entirely. While many LLM providers offer specific frameworks (e.g., the OpenAI Agents SDK or Google Agent SDK), a growing number of powerful open-source, vendor-agnostic solutions are also available, including LangGraph, Pydantic AI, and CrewAI.

To show how the need for manual *glue code* is eliminated, our previous Example 1-1 could be rewritten using Pydantic AI as follows:

Example 1-2. A simple email processor AI Agent with Pydantic AI

```
from pydantic_ai import Agent, RunContext

email_agent = Agent(
    'openai:gpt-4o',
    system_prompt=(
        'Analyse the email and mark it spam or important if it
makes sense'
    ),
)

@email_agent.tool
def mark_as_important():
    # ...
```

```
@email_agent.tool
def mark_as_spam():
    # ...

email_agent.run(email)
```

Graph-based AI Agents

AI agents, in their raw form, autonomously determine on the control-flow and which tools to invoke. While this pattern opens up many powerful new opportunities, it also introduces significant downsides.

Given the inherent unreliability of LLMs, agents don't always choose the most efficient or promising path to achieve their goal. This is particularly true as the complexity of the task increases. Consequently, it has proven highly valuable to define a specific high-level structure within which the agent can operate.

As such, a structured approach leads to significantly more reliable and resilient execution, it is gaining rapid traction across the industry. The established method for achieving this is by defining a graph flow. This defined graph flow is essentially a state machine where the agent's autonomy is intentionally limited to decide only on the state transitions within the pre-defined graph structure.

A good example of this is the *Deep Research* mode popularized by Perplexity and now offered by many other providers. Executing a research task like this, using a raw, unstructured agent, carries the risk of the agent doing either too little research or the agent being trapped in an endless research loop before generating the final report.

As described in Figure 1-4, by defining a graph-flow it can be ensured the agent systematically completes the necessary key stages. Within the structure, the agent still has a certain degree of autonomy and can, for example, decide on the search queries or evaluate if the results have been satisfying.

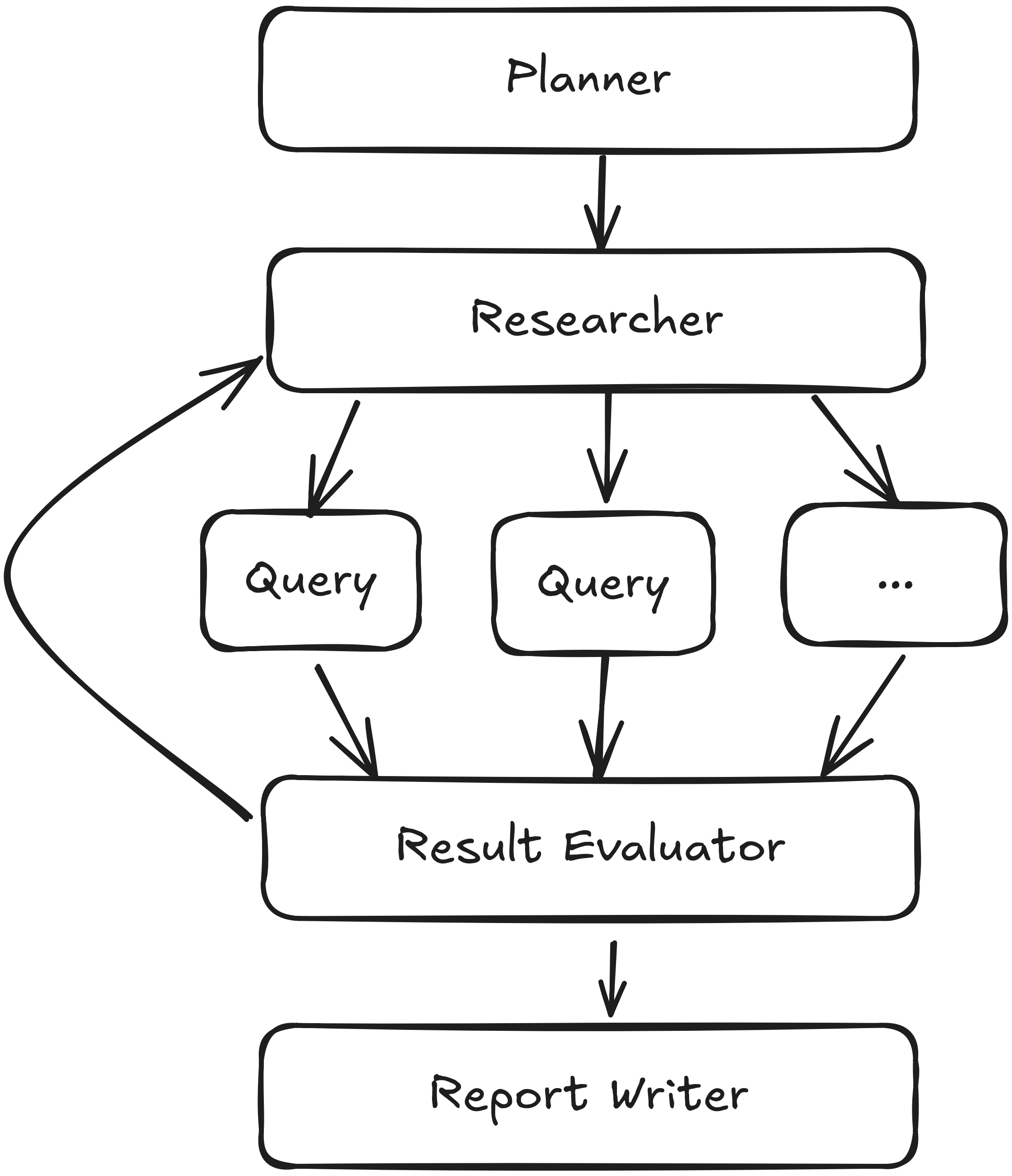


Figure 1-4. Example of a Deep Research Agent

Orchestrated AI Workflows

A workflow is a set of steps that are executed in a specific order. While some, or even every step, might have an Agentic correction loop, this wouldn’t normally be categorised as an Agent.

That doesn’t mean that they aren’t valuable, actually the opposite is true. These workflows are valuable simply because they are the most reliable way to run a set of steps.

Workflows and graph-based control flows are very similar. The only differences are that the control flow is controlled by an external workflow engine rather than within the agent code. This has several major advantages over graph-based control flows because these external workflow engines already have many of the features you need to run these workflows reliably at scale. Features like checkpointing, error handling, parallel execution, retries, monitoring, and many others come out of the box.

An excellent example of this would be an unstructured data extraction workflow using an LLM to extract data from internal docs. This could easily be run in something like Airflow or Argo Workflows simply and safely.

Composite AI Architectures

While these patterns might sound fairly simple, they are still strong building blocks for most AI applications. In practice, however, these patterns are not distinct. Often even the most complex AI applications are still a combination of these fundamental elements. For example, it is very common to have a graph-based agent that implements the RAG pattern in one of its stages and additionally provides a chat interface for user interaction.

Another variation is to have multiple agents collaborating to solve a complex task. For these *multi-agent applications*, the complex task is composed of smaller sub-tasks that are processed by individual optimised agents. For orchestrating multiple agents, several strategies and standardised protocols exist that Chapter 8 explores further.

AI Application Development Lifecycle

Having examined the basic AI application patterns, the next topic is the development lifecycle of an AI application.

Since AI Engineering largely focuses on integrating and utilizing existing foundation models, the necessity for training bespoke models is limited to specialised cases. Consequently, the overlap of AI Engineering with traditional Software Engineering is far greater than its overlap with the core Machine Learning discipline.

While there *are* three or four key differences, the vast majority of the effort is still software engineering. This also becomes imminent when looking at the application development lifecycle outlined in Figure 1-5. Broadly, the cycle follows the same phases as traditional software development, ideally executed incrementally. With only a few exceptions, similar activities are performed in each phase, which will now be explored.

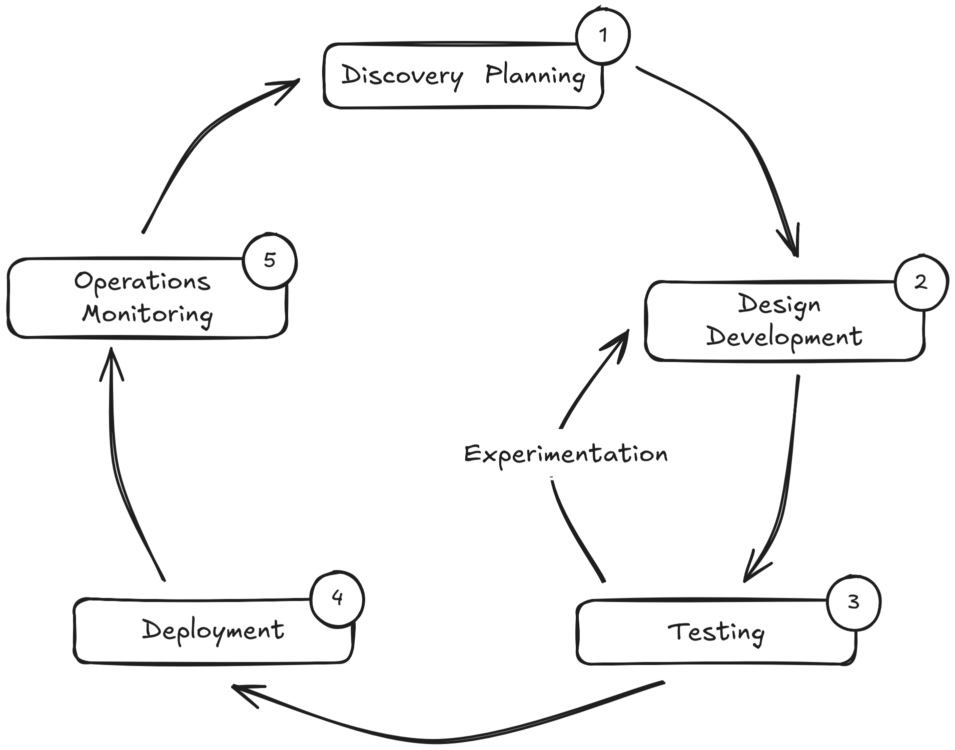


Figure 1-5. The AI Application Development Lifecycle

Discovery and Planning

The *Discovery and Planning* phase initiates the cycle. Unsurprisingly, this is where the purpose and business value of the AI application are defined. Similar to traditional software development, cross-functional requirements must also be collected. Given the non-deterministic and often unpredictable nature of AI systems, it is critical to align early on aspects like compliance, required accuracy targets, and security and safety needs. It is important to remember that the development lifecycle is fundamentally iterative, meaning that the first Minimum Viable Product (MVP) version of the application does not require every detail to be finalized right from the beginning.

Design and Development

Following the planning comes the *Design and Development* phase. As in traditional software engineering, this begins with defining the architecture and design of an application. For AI applications, this includes defining the interfaces for users and other systems, and breaking the solution down into manageable components. Considering AI agents, this includes defining the necessary tools and knowledge bases (RAG sources).

Furthermore, critical technology choices need to be made, such as the programming language, selection of (agent) frameworks, but also the LLM providers including the LLMs to use. On top of writing code, the actual development work also includes prompt engineering, which describes the iterative and experimental process of crafting effective prompts to achieve the desired responses from LLMs.

Testing

The next phase, *Testing*, arguably is the most different from traditional software. Software tests can either pass or fail. Working with AI applications is fundamentally different. The open-ended, non-deterministic nature of the output makes systematic assessment far more challenging, marking this as one of the hardest problems in AI Engineering.

The systematic techniques used to measure the performance, quality, and behavior of the AI component are collectively known as *Evaluation* (or *Evals*), which Chapter 5 explores further. Instead of a simple pass/fail, evaluation typically returns multiple confidence scores across different dimensions (e.g., accuracy, coherence, adherence to safety guidelines).

Due to the black-box nature and complexity of LLMs, the relationship between an implementation change, such as modifying a prompt or changing the model, and the resulting Evaluation scores is often non-obvious. Consequently, achieving better scores requires an experimentation loop (indicated in Figure 1-5) in which developers systematically test numerous potential solutions and iteratively refine their approach based on the evaluation results.

Deployment

The *Deployment* phase is again very similar to established software practices, with minimal AI-specific differences. For example, an AI application can be architected and treated as an independent microservice. This self-contained structure allows it to be deployed and scaled independently. As the AI application matures and moves toward production, standard advanced deployment methods such as (horizontal) autoscaling or canary releases must be considered.

Operations and Monitoring

Once the application is successfully deployed and running in production, the Operations and Monitoring phase becomes important. Initial monitoring begins with operational metrics inherited from traditional software engineering, such as health checks, latency, or resource consumption (e.g., memory). A crucial AI-specific technical metric is the token consumption on LLM APIs, which often represents the largest variable cost of operating an AI application.

While these metrics provide insights on the *technical* health, information on the *business* functionality is required. Independent of the extent of the previous (offline) evaluation, there is no guarantee it covers the real production behavior. Therefore, it is crucial to continuously evaluate the application running in production (often called *online evaluation*). Depending on the available ground truth, this can measure similar dimensions like accuracy, business outcome, or user satisfaction.

These insights allow the development team to iteratively improve the application or to detect and mitigate drift (performance degradation over time).

To increase the explainability and auditability of what are often black-box AI applications, it is an established best practice to record the complete history of internal interactions (e.g., input prompts, model responses, and tool calls). This is most effectively managed via a dedicated tracing system, which Chapter 5 explores further.

Connecting AI Engineering to DevOps

This complete lifecycle, from discovery through operations, mirrors traditional software engineering but introduces new AI-specific complexities at each stage. To manage this new reality effectively and at scale, the industry has naturally adapted the proven principles of DevOps, leading to the emerging discipline of AI Engineering.

The rise of DevOps in the late 2000s fundamentally changed how companies organise software development today. By bridging the gap between *‘development’* and *‘operations’* activities, DevOps provided developers with direct, real-time insights into production environments and, consequently, full ownership over the software application lifecycle.

Through practices like agile methods, cross-functional teams, and Continuous Integration and Continuous Delivery (CI/CD), DevOps empowers teams to autonomously deliver software in small, rapid iteration cycles. This iterative approach not only accelerates development but also allows teams to pivot quickly based on real-time production feedback and business needs. Research has shown that a high DevOps maturity directly correlates with a company’s resilience and ability to navigate in a constantly changing environment. [7]

The DevOps principles have been adopted by the machine learning community under the term *MLOps*. It extends the core DevOps practices to the (traditional) machine learning lifecycle. This includes essential practices around data versioning and modeling, repeatable model training and serving, and continuous model performance measurement.

Now looking into the development of AI applications, do these practices still make sense? This is a clear *yes*, while there are some AI-specific aspects to consider, these practices actually might be more important than ever before. The challenges in software development that DevOps addresses are mirrored in the development of AI applications. In fact, many are even more pronounced.

Everyone has seen the disruptions the general availability of LLMs has caused, and the rapid pace in which new trends are created. While of course not each trend needs to be blindly followed, many of the changes in the Generative AI ecosystem can impact the AI development fundamentally. As an example, it might happen that a new generation of an LLM gets released that is for example cheaper or has a lower latency - better suitable for an AI development team’s use case. In such a situation, the AI development should be to make the decision of using this new model independently without a lengthy coordinate coordination across the company. Generally, in a highly dynamic environment such as the AI ecosystem, it is essential for teams to be able to react autonomously to new developments and change their applications accordingly.

Additionally, the non-deterministic nature of AI applications makes it hard to predict the full extent of an application’s behavior in production and with it the actual business value it creates. As a result, development teams need to get as many production insights as possible so that they can react to it and integrate this feedback into their AI application.

In the current dynamic field of AI, terms are often a bit blurry and used inconsistently. While not directly in the tradition of DevOps, the term that best describes the discipline of developing an AI application throughout its entire lifecycle is *AI Engineering*. A concept that gained significant establishment and traction through the publication of Chip Huyen’s foundational book of the same name.:[8]

While MLOps focuses heavily on the training and operationalization of predictive models, AI Engineering expands this scope to focus on building AI applications, often by leveraging foundation models as described in the previous section. Although the more specific operational term GenAIOps exists, it has not yet gained the same broad recognition, making AI Engineering the more suitable comprehensive term.

Following the DevOps philosophy, it’s best practice to make an autonomous team responsible for the entire lifecycle of an AI application. However, managing this complexity is a significant challenge, as AI Engineering demands expertise across data engineering, AI modeling, software engineering, and DevOps practices. Building a golden path that works for the wide variety of users and use cases is a challenge that is not easy to solve. This is why sensible defaults and a central platform offering are so important. It allows users to focus on the core value proposition of their application, while the platform takes care of the rest.

AI Applications vs Microservices

Considering their base architecture patterns and their lifecycle, AI applications are not that different from traditional software. To tease out these similarities and differences further, it is necessary to compare AI applications (keeping workflows aside) to microservices, as shown on Figure 1-6.

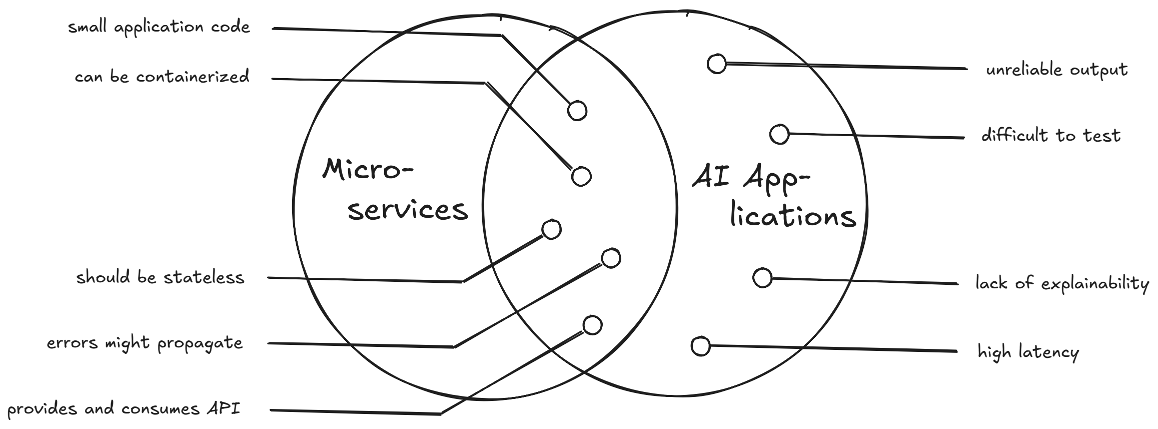


Figure 1-6. AI Applications vs Microservices

Similarities

Microservices are an independently deployable unit that model a specific business capability. And in this regard, AI applications are no different.[9]

Both microservices and AI applications are of manageable size. When implemented in a stateless manner, they can be containerised easily and deployed independently.

Additionally, microservices typically expose a defined API interface (e.g., RESTful) and consume other services via APIs. AI applications are no different. Even their basic building block, the foundational models, are typically accessed via defined API interfaces.

A well-known challenge of microservices is that errors might propagate through the system. Often addressed with circuit breakers, retries, and fallback strategies. AI applications face the same challenge, but as the quality of their answers might vary significantly, this challenge is even more pronounced and needs to be addressed similarly.

AI-specific characteristics

While not strictly incompatible with the definition of microservices, AI applications have some unique characteristics that differentiate them from typical microservices. First of all, as they are inherently non-deterministic and probabilistic, they produce an *unreliable output*. A characteristic that each software engineer would normally avoid at all costs.

From a business perspective, this core characteristic is challenging in itself. How does the business value of an application look like when the accuracy varies? What are the safety implications of an application that might hallucinate? And additionally, how can these aspects be planned for in advance when evaluating the business case of an AI application?

Additionally, from a technical perspective, this unreliability introduces consequential problems. As discussed earlier, AI applications are hard to test and need to be evaluated instead. Considering that AI agents autonomously decide on their control flow, this makes it even harder to predict, debug, and explain their behavior. Undefined behavior might not only lead to wrong answers, but also to security risks, compliance violations, or increased costs.

There is also the issue of managing the large *black-box* that is an LLM. The high release frequency of new LLMs, makes it hard to keep track of the changes and to ensure that the application is not impacted. Tuning an AI application to a specific model does not guarantee that it will perform well on a different model. Different models cannot be treated as simple drop-in replacements due to variations in performance, cost, and API signatures. You can also tune the model to a specific task, but you may then lose its generic capabilities.

Furthermore, AI applications typically have different performance characteristics to microservices. Especially, the latency of LLM calls is significantly higher than typical API calls.

AI-specific challenges

Having identified the unique challenges of AI Engineering, we will outline established strategies to address them. These strategies are crucial from a platform perspective, because the platform must provide the necessary tooling and services to facilitate them. Consequently, we’ll explore them and the necessary platform offering in greater detail in later chapters of the book.

Difficulty in Articulating Business Value

Predicting the performance and eventual accuracy of a novel AI application upfront is inherently challenging. This technical uncertainty makes it difficult to articulate and quantify the final business value and return on investment (ROI) before the development cycle begins.

To overcome this core uncertainty, it is essential to validate the central business hypothesis as early as possible in the development lifecycle. Again, established practices from software engineering can be leveraged. By using techniques like rapid prototyping and building MVPs, early versions of the application can be released to real users early on. In this initial phase, the focus should be purely on gathering real user feedback to validate the central hypothesis. It is essential to formulate clear hypotheses about the expected business value and define measurable success criteria, e.g. time savings on an AI-accelerated process.

In this early phase, there is no need to optimize prematurely for costs (e.g., using cheaper foundation models) or maximum accuracy. By restricting the first release to a small (possibly internal) audience, the impact of any undesired behavior can be minimized.

Imperfect accuracy

The inherent unreliability of AI applications remains the core challenge. While there are technical strategies to mitigate this, it is essential to accept that AI applications will never be perfect. Therefore, this is something that needs to be addressed holistically, also involving the business stakeholders. Depending on the use case, for example, this might be accepting the (financial) risk of potential errors or by including human review steps (*human-in-the–loop*) into the user journey. Considering an agent that helps in planning travels, it probably is a good idea to ask the user to confirm the final plan before executing any booking.

This of course does not mean that technical strategies to improve accuracy should be neglected. To ensure that AI applications align with the intended behavior and don’t provide harmful content, it is essential to implement *guardrails*. Guardrails describe policies and automated checks that restrict the AI application’s behavior to acceptable boundaries. These checks can consist of traditional software checks but also of additional LLM calls that verify the output against defined guidelines.

To improve the accuracy of the AI application, the typical first step is *prompt engineering*. This technique describes experimenting with different prompt formulations to achieve better results from the LLM.

When knowledge in a specific domain or recent information is required, *RAG*, which was introduced earlier, is an established technique by providing the necessary information to the LLM at runtime.

And finally, for applications executing highly specific tasks, *fine-tuning an LLM* on domain-specific data can be an additional strategy. However, as this requires significant expertise and resources, it should only be considered when other strategies are insufficient.

Challenges in Testing

Applying all the strategies to improve accuracy makes sense if their impact can be measured reliably. As outlined earlier, the open-ended output makes traditional software tests not suitable. Instead, the application needs to be *evaluated* systematically.

This requires defining an evaluation dataset with example inputs and expected responses. During the evaluation, the actual application output is compared against the expected responses using defined metrics, that need to be aligned with the business goals.

A key best practice is *Evaluation Driven Development*, which describes defining the evaluation criteria upfront and using them to guide the development process. This ensures that every change made to the application is directly assessed based on its impact on the defined metrics and avoids aimless, time-consuming development which at the end does not improve the application.

Explainability and auditability

While evaluations are essential, they alone are not sufficient to provide insights on the AI application’s behavior in production. The real system inputs might differ significantly from the evaluation dataset, and consequently, it is essential to monitor and gather insights on the real production behavior.

Besides *logging*, the established best practice to achieve this is to implement a dedicated *Tracing* system, which records all internal interactions of the AI application (e.g., input prompts, model responses, and tool calls). This helps in debugging issues, understanding real user behavior, and storing necessary information for compliance and audit purposes.

Rapidly evolving ecosystem

As AI Engineering is a young discipline, the ecosystem is still evolving rapidly. New LLMs, frameworks, and best practices are emerging constantly, and technology choices made today might not make sense in a few months’ time.

Therefore, for teams building AI applications, it is essential to *embrace change*. A *modular application design* that allows swapping components (e.g., LLM providers) easily is a key best practice. Furthermore, *team autonomy* as discussed in Section “Connecting AI Engineering to DevOps” is crucial, as it allows teams to react quickly to new developments without lengthy coordination.

Anti-patterns in the Application Development Lifecycle

As with any new Technology, there is a lot of hype and misconceptions. In practice, the presented patterns and strategies are often wrongly applied. To strengthen the understanding of them, this section contrasts them with commonly observed anti-patterns.

AI for AI’s Sake

People being exposed for the first time to the powerful capabilities of LLMs often fall into the trap of using them for everything. In practice, prompts are often used where if statements would suffice. LLMs are sometimes used for expanding acronyms when text expansion would give perfect accuracy.

As a simple rule of thumb, if you can write deterministic software to solve the problem, do it. This avoids all the challenges of AI applications and often leads to better, safer, and cheaper software.

> *It is tempting, if the only tool you have is a hammer, to treat everything as if it were a nail.*
>
> —Abraham Maslow

Reimplementing orchestration within applications

Orchestrating a deterministic set of tasks is a solved problem. Many cases exist where teams try to reimplement orchestration within the applications, or worse within a non-deterministic agent.

If you need to orchestrate multiple tasks following a fixed order, look first into existing workflow or pipeline orchestration engines, as they provide a lot of capabilities that you will need sooner or later out of the box. In other words, if the control flow is strictly deterministic and follows a fixed order, you should favor the “Orchestrated AI Workflows” pattern. However, for scenarios requiring guided, LLM-driven decision-making within a structured flow, “Graph-based AI Agents” are the appropriate choice..

The *God* Agent

Many AI enthusiasts dream of building a single *God* agent that can solve any problem given to it. This is a seductive concept, and maybe will work at some point in the future. However, for now (in 2026) this approach typically produces poor results. Any sufficiently valuable domain to build an application in is likely to be too complex for a single *God* application to effectively choose the most correct action.

For example, a single application tasked with handling the all enterprise functions from HR to logistics would struggle to choose the correct actions efficiently.

For now, a much more effective approach is to use a combination of applications, tools, and techniques to build smaller specialised subsystems. This way, the unavoidable error cases can be brought under control much more effectively.

> *One application to rule them all, One application to find them, One application to bring them all, And in the darkness bind them*
>
> —J R R Tolkien

The applications that should just be a tool

At the other end of the spectrum, AI agents are often built for every small task. Often it is much more efficient to just build a simple tool that does one thing well and which then can be used by higher-level agents. If this tool needs to be shared within an organisation, then building it as a Model Context Protocol (MCP) server makes a lot of sense, which Chapter 8 further explores.

> *Stop trying to make fetch happen!*
>
> —Regina George

Technology-Driven (Not Problem-Driven) Development

As sure as night follows day, with each new technology craze people will try to integrate it into every application. For example, Multi-agent systems or graph-based agents are very powerful and exciting patterns, but they are only valuable for specific use cases. Implementing such a system is not a goal in itself and comes with significant complexity and overhead.

While exploring new technologies is encouraged, always keep the end goal in mind. Explore them in small isolated experiments first before integrating them into large, real applications. This anti-pattern is especially dangerous when these applications become part of larger roadmaps and people only question their value months or even years down the line.

Focus on the real value.

The Chatbot-Only Mindset

While chatbots are a very popular interface of LLMs, they are by no means the only one. In fact, the more and more complex applications become, the less likely it is that a chatbot interface is the best way to interact with them.

Instead, in the near future, the majority of AI interactions will happen under the surface. Which requires deep integration into existing user-journeys with innovative ideas on the user experience.

Aimless development without Evals

One of the major issues with AI development is that you need to have an incredibly nuanced opinion on quality aligned with the business goal. More often than not, you have a huge potential space of inputs and outputs, and it isn’t always clear whether you are improving things. When everything is changing all the time, it’s very easy to drift along without any direction, just coasting on vibes. This is clearly not ideal, so bring in evaluations as early as possible. Ideally, you begin with the end in mind and do *Evaluation Driven Development*, which means defining your evaluation criteria upfront before writing any code.

Analysis Paralysis from Evaluation Challenges

At the opposite end of the spectrum is delaying the release of your application because you can’t perfectly evaluate it yet. Evaluating AI applications is admittedly difficult, especially at the start. Striving for perfection brings a huge risk of just wasting resources and money. No evaluation approach can guarantee that your application will provide real business value.

Imagine a team that builds an internal summarization tool. They might define an evaluation dataset centered around summary quality metrics like coherence and relevance. However, after the initial release they learn that their users actually wanted longer summaries.

Make sure incomplete evaluation does not paralyse you and aim for releasing early. Do it in a safe way though, either by rolling it out to a small user group or by providing it as an optional add-on. As soon as you collect the first user feedback, you can use this to improve upon your evaluation and make your baseline for future iterations more solid.

Conclusion

In this chapter, a comprehensive overview of AI applications and their development lifecycle has been provided. When building an AI application platform, this understanding helps to derive the most critical capabilities the platform must provide.

The first apparent theme of this chapter is that there is a significant overlap with traditional software development. For platform teams, this is good news, as it means there is no need to reinvent the wheel when aiming to support AI Engineering activities. Instead, they can strategically leverage already available capabilities and tooling from existing internal development platforms (IDPs).

Naturally, there are some crucial AI-specific aspects that need to be considered. The application patterns help to understand how AI applications are typically structured and which building blocks are commonly used and therefore need to be provided by the platform. This, for example, includes integrating LLM APIs, enabling RAG capabilities, and providing workflow or agent orchestration tools.

The application lifecycle clarifies that AI Engineering requires much more than just integrating LLMs into applications. And here the real value of a dedicated AI platform becomes apparent, as it must support AI developers throughout all challenges they face during the entire lifecycle of their application, without restricting their autonomy. For a platform, this means that the necessary tools and services (e.g., for executing and tracking evaluations, or for operating AI applications) must be provided so that developers can choose the best fitting ones depending on their specific needs.

Additionally, we have to take into account that AI Engineering is a relatively young discipline. Many enterprises will only have a limited pool of experts capable of handling the broad scope of this discipline. If a platform is seen as an accelerator, it can contribute to upskilling an organisation and ensure expertise about AI Engineering is shared. This can, for example, be achieved by providing sensible defaults (e.g., recommended technology choices), and pre-configured building blocks, which significantly lowers the entry barrier for teams venturing into AI application development.

The next chapter will introduce the concept of platforms and their value proposition. After that, the following chapters will explore the necessary platform capabilities in greater detail, to support and accelerate building AI applications effectively at scale.

---

[1] And naturally those coding assistants can be used to write AI applications. Are you dizzy already? This topic is explored in Chapter 12 in more depth.

[2] One of the authors by the way

[3] Andrej Karpathy introducing the term *Vibe Coding* on X

[4] Kent Beck in The Pragmatic Engineer podcast

[5] No language models were harmed (or used) in generating the text of this book. However, AI was consulted during quality control to make us look good to our editor.

[6] Instruction fine-tuning is a specialised post-training process in which a pretrained LLM gets further trained on following human instruction and generating corresponding responses. This is achieved by providing datasets with both instructions and related responses.

[7] A key measure for this maturity is the set of DevOps Research and Assessment (DORA) metrics. Nicole Forsgren, Jez Humble, and Gene Kim have statistically analysed in Accelerate how those are a leading indicator for organisational performance.

[8] AI Engineering by Chip Huyen, O'Reilly Media, 2023, *https://www.oreilly.com/library/view/ai-engineering/9781098166298/*

[9] Admittedly, it is not strictly necessary for an AI application to be a separately deployable unit. However, encapsulating them well from the rest of the greater application is recommended, though this can also be achieved by following different approaches, such as a modulith.

Chapter 2. What Is a Platform?

"Improving daily work is even more important than doing daily work."

—Gene Kim, The Phoenix Project

The previous chapter has shown that the DevOps principle *You build it, you run it* is as important to follow for AI applications. This means teams have to take on a lot of responsibilities, leading to a higher team cognitive load — the total mental effort required to build and run an application. Unmanaged, this cognitive load slows down value delivery.

This problem is not unique to the relatively young field of AI Engineering and has been recognised for some time in the software engineering community. To address this challenge, the term *Platform Engineering* has been coined to describe the idea of providing centralised services and tools, ultimately reducing the cognitive load on development teams.

This idea is not new and probably dates back to the early days of IT. But with the rise of cloud computing, DevOps practices, and the need for faster software delivery, platform engineering has gained traction in recent years and became a recognised discipline with its own best practices and patterns.

A key concept is treating the *Platform as a Product*, which emphasises the need for platforms to be designed, built, and maintained with the same care and attention as any other product.

Platforms, like DevOps, are also an organisational pattern. The book Team Topologies[1] is often seen as a foundational reference for organising and dividing teams efficiently in modern organisations. It formalises the challenge of cognitive load for teams and its impact on delivery. The book describes how a dedicated Platform Team works to reduce the cognitive load on Stream-Aligned Teams (those focused on delivering customer value) by providing internal services, thereby enabling faster and safer software delivery.

You might wonder with the extensive offering of cloud providers, why is there a need for internal platforms at all? As described in Effective Platform Engineering [2], the cloud provides low-level, general-purpose building blocks rather than an out-of-the-box solution. Platform Engineering is about closing the gap with services that the organisation needs for efficient delivery. This is also true for AI applications. While there are, generally applicable patterns, which are presented throughout this book, building an effective platform highly depends on the context of your organisation.

This chapter provides a foundational understanding of platforms, and how AI Engineering platforms fit into the broader platform ecosystem within an organisation. It describes the key success factors for building and maintaining effective platforms, as well as common anti-patterns to avoid.

The Platform Definition

Let’s start with defining what a platform is. Evan Bottcher, a leading voice in modern software architecture best known for defining the digital platform as a compelling internal product, puts it this way:

A digital platform is a foundation of self-service APIs, tools, services, knowledge and support which are arranged as a compelling internal product. Autonomous delivery teams can make use of the platform to deliver product features at a higher pace, with reduced coordination[3].

This definition highlights several important aspects of a platform. Firstly, a platform is more than just infrastructure - it's a foundation that enables teams to deliver value autonomously with minimal coordination overhead.[4] It combines technical capabilities with knowledge, support, and organisational patterns. Taken to its logical extreme, a platform's simplest form is a well-documented set of sensible practices. In reality, most platforms will include a combination of services, tools, and Application Programming Interfaces (APIs).

The next key aspect is *self-service*. This includes easy access to the platform capabilities, as well as the ability to provision, configure, and manage resources without needing to go through a central team or approval process. On the one hand, this is important for the platform's users - the stream-aligned teams. They need to be able to move fast and make decisions autonomously, without being blocked by the platform team priorities and roadmap ("Backlog Coupling"). On the other hand, this is important for the platform team itself. For them to be effective, they need to focus on building and maintaining the platform capabilities, rather than being slowed down by requests and approvals from individual teams.

Additionally, a platform should be treated as a *compelling internal product*.[5] Platforms succeed when they're treated as products that must win adoption through value, not mandate. This requires the platform team to focus on understanding the needs of their users - identifying the features providing the most value, and continuously improving the platform based on feedback. While this sounds simple, it requires organisational changes beyond technology and includes redistributing operational responsibilities.

Finally, the definition closes by highlighting the central goal of a platform: accelerating the development and delivery of applications by ensuring real team autonomy.

Platform Anti-Patterns

Before diving into what a good platform looks like, let’s look at some of the common platform anti-patterns that are often seen in the wild. These will help you to understand the challenges of building a platform and how to overcome them. You may recognise some of these patterns from your own experiences.

The Golden Cage

The *golden cage* occurs when the *golden path* for building on the platform doesn’t allow for any deviation from the platform’s core capabilities. This happens when platforms impose a rigid *golden path* without allowing for necessary deviations, effectively preventing value stream teams from innovating or solving problems in ways the platform didn’t anticipate.

Enforcing Mandates instead of Winning Adoption

Unfortunately, this is something that happens all too often. If you need to mandate a feature instead of winning adoption, then you are likely missing some key user needs. Sometimes mandates can be used to reduce the cognitive load of the platform, but they often mask poor product decisions. Investigate the user needs and see if there are any gaps in the platform that can be filled. Platforms rarely get enough user feedback to easily determine what they need to improve. When you are getting this feedback, you need to listen to it and make changes.

Self-service with Approval

Users should be able to self-service the vast majority of their needs. This means that there should be no need for approvals or approval processes. Every approval slows down the ability of stream-aligned teams to deliver value.

Obscure *Self-service*

Another anti-pattern for self-service is when it is so complex, poorly documented, or unintuitive that users struggle to find or use the self-service capabilities effectively. This can also happen when the underlying pipelines take too long or are flaky. In these cases, the self-service capabilities are not effective and users must use the platform as a black box.

Backlog Coupling

Coupled backlogs are a common anti-pattern in software development, not just in the context of platforms. Platforms should aim to reduce the coupling between backlogs. The clearest way to do this is to ensure that stream-aligned teams are not constrained by the platform's backlogs.

A Separate *Special* Runtime for AI Applications

Given that AI applications behave like services calling a Large Language Model (LLM), why is a separate runtime needed for AI applications? This is another example of reinventing the wheel for applications. There is no compelling reason for this, but there are downsides. Firstly, it is a significant amount of work to build and maintain a runtime for applications. More importantly, it forces developers to learn a new set of sensible defaults, none of which help them deliver faster. In the worst case, if you use all the same sensible defaults, then you have completely duplicated the existing runtime, so you might as well have just used that!

Over-Focusing on Prompt Management

Many organisations focus on sharing prompts within the whole organisation.

But in practice, prompts have limited reusability. Newer models are much more *robust* to prompt perturbation (i.e. small changes in wording), reducing the need for highly optimised prompts. Additionally, they are just text and tailored to a very specific use case and maybe even to a specific

LLM. Prompts may stop working effectively with a new LLM version or a different provider.

The reality is that very few people will use your perfect prompts. Therefore, building complex infrastructure to edit, share, and version them is considered an anti-pattern.

Instead, keep the prompts in version control systems close to the code that calls them. Feel free to make them variables and put them in separate files, which can allow you to do all the A/B testing you require.

This does mean that non-technical people cannot edit them, but if the technical barrier is editing a text file in a GitHub User Interface (UI), then organisations should be able to upskill people relatively easily. This can dramatically simplify what you need to build and maintain as a platform. It will also help teams with debugging and troubleshooting.

One exception to this is providing prompt hints to users so that they can get started. Doing this with a Model Context Protocol (MCP) server is relatively trivial.

Delegating Complexity to Users

This is, unfortunately, a common anti-pattern. Fundamentally, the most valuable thing a platform can do is reduce the cognitive load for the stream-aligned teams. If the platform team is not able to do that, then it’s not a platform.

In large enterprises, this complexity may come from many different sources. The temptation to delegate complex tasks to the user because it’s too complex for the platform team is very strong.

Using a Platform to Hide Organisational Issues

Another common anti-pattern is to paper over the real problems of the organisation. A good example would be data governance that locks down metadata about data products. This means that discovery and access to data products are not possible without the proper permissions. You could have a solution that allows users of your platform to discover data products with your elevated permissions, but this is clearly subverting the data governance.

The real problem is that data governance is not fit for purpose. Rather than passing this responsibility to the users of your platform, focus on lobbying for better sensible defaults that benefit the entire organisation, not just the Data Governance team.

Reimplementing Existing Tools and Frameworks

Reimplementation has a lot to do with prioritisation; yes, you could probably improve the existing tools and frameworks, but it's not the right thing to do.

Your first step should be to identify the real problem that the existing tools and frameworks are trying to solve. You should also use the tools and frameworks available to you.

Only look at reimplementing existing tools and frameworks if you come across a major gap in the existing offering. If you are able to improve the current offering, you should do that.

Success Factors for a Platform

Those are my principles, and if you don't like them… well, I have others.

—Groucho Marx

As outlined in the preceding section, building an effective platform requires a product mindset. Building a product is hard, and there are many books and resources available on this topic, so this chapter cannot provide an exhaustive overview of strategies.

However, there are some specific success factors that are particularly important for platforms, which the following sections outline.

Adopting a Product-Based Funding Model

If the organisation is unwilling or unable to invest in a product team for a platform, then unfortunately the platform is doomed from the start. One of the well-known facts about technology is that it's hard to predict how it will evolve. Without a product team, the platform will be stuck in a state of *legacy* and will not be able to grow.

One of the best ways to change this mindset is to highlight some of the benefits of a platform to Leadership.

Identifying Core Platform Capabilities

When defining the platform's capabilities, the key is optimising for global efficiency rather than local optimisation of individual teams or functions. A different way to think about this is the well-known Pareto principle (80/20 rule). Freely adapted to platforms, this means: by focusing on the 20% of needed capabilities that are used by 80% of teams, platforms can provide significant value while minimising complexity and maintenance overhead.

So how does one identify this 20%? First, this requires a deep understanding of the needs of the stream-aligned teams. However, teams are often unable to articulate clear requirements upfront. This is frequently due to the *XY Problem* [6]. Teams get fixated on a solution they've already conceived, which prevents them from stepping back and articulating their true underlying problems. The relative youth of the AI Engineering discipline, lacking many widely established best practices, further contributes to this challenge, as teams might not yet know what *good* looks like. Therefore, implementing what they ask for poses the risk of curing symptoms instead of addressing the underlying needs.[7] Additionally, it might lead to addressing only the requirements of the most vocal teams, instead of finding solutions that work for 80% of teams. To address this, it is essential to be empathic: dive into their workflows and pain points to identify the real underlying challenges.

Secondly, developing an understanding of common patterns and approaches across the AI Engineering discipline is key. This not only helps to understand and validate the requirements raised by the users but, critically, it allows the platform to drive sensible, efficient defaults across the entire organisation. This very book is designed to guide you there. [Link to Come] outlines the common building blocks for AI platforms, and subsequent chapters dive deeper into specific capabilities.

Thirdly, the platform must support a variety of use cases. From simple chatbot prototypes to complex, highly available multi-agent systems. Additionally, teams will have varying levels of expertise, ranging from AI novices calling an LLM for the first time, to AI experts who have fine-tuned LLMs already. A good starting point to balance these interests is the concept of 'Designing for extreme users'[8] from Design Thinking.

As outlined in Figure 2-1, the idea is to gather the requirements of the platform not for the majority of average use cases and users, but instead to focus on the extremes: the most simple and the most complicated use cases - and equivalently - the novices and the expert users. The idea is that by addressing their needs, the platform will also suit the average users in between and allows for a growth path when use cases and user expertise evolve over time.

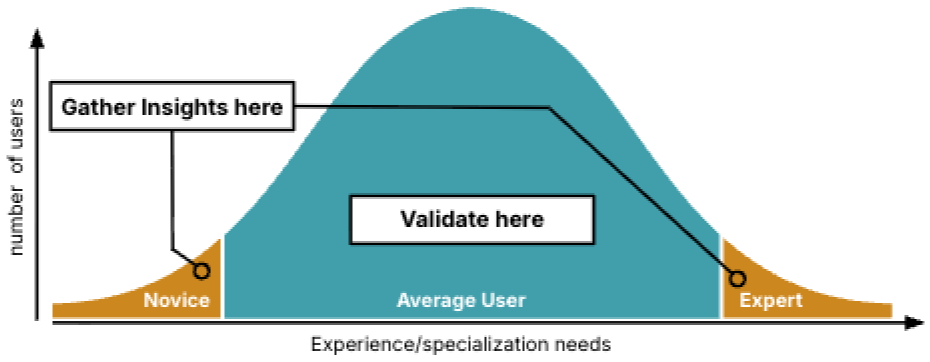


Figure 2-1. Designing for Extreme Users

Finally, as with any other software product, it's important to start small and iterate based on feedback. Identify the most critical capabilities, build a minimum viable platform, and then collect feedback from your users and use it to improve the platform.

Over time, this will expand the platform capabilities to a point where you have an extensive catalogue of services, tools, and documentation.

Integrating Capabilities Effectively

Just identifying and providing those necessary capabilities is not enough. Such a platform quickly devolves into a fragmented collection of guides, tools, and services. Ultimately, this forces teams to waste time manually *stitching together* components.

Therefore, for a platform to be truly effective, it must go one step further and optimise the capabilities for genuine interoperability and composability. Ensuring so that teams can easily combine components without incurring significant integration overhead. This involves foundational, platform-wide standards. For instance, all provided services should follow consistent authentication and authorization mechanisms and log to a unified observability backend. But this also requires ensuring that components are composable at runtime and can easily access each other. For example, the platform's agent runtime environment should seamlessly allow access to the platform's vector database service. This can be achieved through mechanisms like automatically injecting connection details and providing pre-configured default permissions.

Having platform components readily available for effective integration is called the *mise-en-place* principle, visualised in Figure 2-2.

Figure 2-2. Mise-en-place principle

This principle is borrowed from the culinary world. It originated within the French cuisine and translates to *everything in its place*. It describes chefs preparing and organizing all their ingredients and tools before starting to cook. This includes chopping vegetables, measuring spices, and arranging utensils. Following this metaphor, that is what a platform should strive for. All components should not only be available, but also organized and arranged in a way that makes building applications efficient. Not each dish needs all ingredients, but having them ready and available makes the cooking process much smoother and faster. That’s not different for AI platforms. Not every team will need all capabilities, but an offering that covers the whole lifecycle and the needs of different application types will allow teams to pick and choose what they need in their current context, while always having the option to extend and grow into other capabilities over time.

Avoiding Technology Lock-in

As described in the previous chapter, an additional challenge is the fast evolving and highly dynamic nature of the AI Engineering ecosystem. A tool or framework that is popular today might be completely obsolete in a few months. This poses a crucial challenge for AI platforms, because as soon as users rely on a specific tool or framework, it becomes very hard to change or remove it without breaking existing applications.

While this probably cannot be completely avoided, there are some strategies to mitigate this risk. Avoid vendor lock-in by leveraging open standards such as OpenTelemetry for observability (more about this in Chapter 7), or using the MCP for providing agent tools (see Chapter 10).

An additional strategy is JJeff Bezos's, founder of Amazon, *two-way door* principle[9]. This principle distinguishes between decisions that are irreversible (one-way doors) and those that are reversible (two-way doors). This means whenever possible reversible decisions should be preferred, and they should be made fast. Any *failures* from fast, reversible decisions are valuable learning opportunities in the innovation process.

While a certain degree of abstraction clearly helps to decouple the platform from specific tools or frameworks, this is an approach that should be followed with care. The risk is that this abstraction becomes - what Gregor Hohpe, an influential author and Enterprise Strategist known for his book Platform Strategy, calls a *Grim Wrapper* [10] - an abstraction that fails to completely hide the underlying complexity and cannot keep up with the evolution of the underlying tools or services. Internal maintained abstractions will likely never be as well maintained, documented or feature rich as the underlying tools or frameworks. To avoid making the platform the bottleneck for innovations, it is recommended to focus on providing sensible defaults and easy on-ramps for popular tools and frameworks, rather than trying to abstract them away completely.

For example, some platforms try to abstract away specific LLM providers behind a common interface for central governance reasons. But in practice that quickly became a leaky abstraction as each provider has its own unique features. Considering, that most (open source) LLM-related libraries and agent frameworks already provide abstractions for multiple LLM providers, this just added unnecessary complexity for users, because the internal abstraction was not directly compatible.

TIP

Besides the actual capabilities of the platform, the most crucial factor contributing to the success of one is the developer experience (DevEx).

DevEx, describes the overall experience and satisfaction throughout the full application lifecycle of developers when interacting with the platform. Imagine, you have the most powerful platform with all the capabilities you could wish for, but nobody engages with it because it’s hard to use, poorly documented, or just unreliable. This would make the platform useless and a waste of the investment. Therefore, it is essential to focus not only capabilities, but also ensure a great DevEx.

Comprehensive Documentation

Starting with the most obvious aspect of DevEx, the capabilities and usage of the platform should be well documented. And no doubt, this is easier said than done. The platform is often developed by engineers, and documentation is often seen as a secondary task. Additionally, as outlined in the previous chapter, AI Engineering is a young discipline with many engineers being new to the field. This means the documentation needs to be targeted to a diverse audience with varying levels of expertise.

One approach to address these challenges and to allow users to navigate based on their needs is to structure the documentation following the Diátaxis framework.[11] This framework advocates for organizing the documentation into four different types: *tutorials*, *how-to guides*, *reference*, and *explanation*. Tutorials are step-by-step guides for beginners to get started with the platform. In contrast, how-to guides outline for already-experienced users how to perform certain tasks. References are for detailed, comprehensive information about the platform’s capabilities and includes API definitions and configuration options. They are not meant to be read linearly, but rather to be used as a map for looking up specific information. Finally, explanations provide the underlying concepts and principles behind the platform.

Breaking this down for an AI platform, building and deploying a simple AI agent could be a tutorial. Explaining how to set up a vector database integration could be a how-to guide. The API reference for the platform's software development kit (SDK) would be located in the reference documentation. And finally, an explanation could cover the architecture and design principles of the platform or why a certain technology (e.g., why a vector database was set as the platform's default).

Besides structuring the documentation effectively, it's also important to ensure that the documentation is up-to-date and easily accessible. A well-established practice is Docs as Code. [12] This approach keeps documentation close to the code and manages it with the same tools and processes as the code itself. This encourages developers to update the documentation as part of their regular workflow, ensuring that it stays current and accurate. A further extension of this idea is to generate documentation directly from the code (e.g., from Python docstrings for software SDKs) or to generate code samples from tests.[13] This guarantees that the documentation is always in sync with the actual implementation.

Especially when the platform is developed by multiple teams or when there are multiple (integrated) platforms in the organisation, it's important to make sure the documentation is hosted in a central portal and is easily searchable. This does not only help users to find the information they need quickly, but also lays a good foundation for using AI-based tools making the documentation even more accessible, which will be covered in Chapter 12.

Intuitive API design

An intuitive API design is even better than good documentation. If the APIs are designed following clear conventions, users can often figure out how to use them without having to read the documentation.

For platforms this is not an easy task, as they often don't expose a single API - but rather multiple across different technologies. This can include SDKs, command line interfaces (CLIs), configuration files, REST APIs and more. First and foremost, it's important to adhere to the conventions within each technology. For example, SDKs should follow the conventions of the programming language they are written in (e.g., `snake_case` function names for Python) and CLIs should follow common patterns for command-line tools (e.g., a `--help` command). Beyond that, there are some general best practices, such as input validation and clear error messages that naturally should be applied across all APIs.

The key to good API design, however, is to think from the user's perspective. The majority of users want to accomplish specific tasks, and APIs should be designed to make these tasks as easy and intuitive as possible. Most users don't need to understand all the implementation details of the platform - they just want to get their job done. And this is what good APIs do: they hide the internal complexity and provide a well-abstracted interface for users to interact with. An effective way to achieve this is to identify the most common user journeys and optimise the APIs for these journeys. This includes providing sensible default values to reduce the number of required parameters, because there is nothing more frustrating than having to provide a dozen parameters which you don't yet fully understand just to get started.

For example, if the platform allows deploying an agent through an SDK, it is helpful to enable this with a default resource configuration (e.g., memory) balancing costs and providing sufficient resources.

Team Autonomy and Sharp Tools

While optimising APIs for the most common user journey is important, there is a risk of creating a *golden cage* that prevents teams from innovating or solving problems in ways the platform didn't anticipate.

This especially happens when platforms don't have full insights into the diversity of their use cases and try to enforce too many prescriptive conventions or rigid mandates. The tricky reality is that various teams have distinct needs and might deviate from the platform's *sensible default* setup in different areas. In practice, this is often the biggest blocker to platform adoption. Teams feel restricted by the platform offering and, instead of benefiting, they feel slowed down by fighting the platform's limitations, which would be the exact opposite of the platform’s goal to enable team autonomy and speed up delivery.

In terms of API design, Carson Gross, the creator of htmx, advocates for a layered API design with *‘escape hatches’* allowing users to access the next lower abstraction layer.[14] This allows users to start with the high-level abstractions for common use cases, but also provides expert users with the flexibility to use *sharp tools* and deviate when needed.

Applied to platforms, these escape hatches should allow teams to override the provided sensible defaults built into the API, but even one step further. It should also allow teams to deviate from the platform’s tool and service selection.

One example could be a platform running on a cloud provider, which integrates a subset of services natively into the platform (e.g., by provisioning default permissions and configuration). An escape hatch would be to allow users to access other services of the underlying cloud platform by provisioning them and defining permissions themselves.

Referencing the earlier discussed *mise-en-place* principle, this means a platform should not hinder teams from bringing their own ingredients and tools. Naturally, teams then have to prepare (configure, integrate, operate) those ingredients themselves, but they are able to do so and can cook a meal not foreseen by the platform.

Or as Gregor Hohpe summarizes:

> *If your users haven’t built something that surprised you, you probably didn’t build a platform*[15].

Additionally, such openness helps in navigating the fast-evolving AI ecosystem. It’s unlikely that a platform team can keep up with all the latest tools and frameworks, but if the platform is open and even encourages experimentation, individual teams might be able to experiment with new tools and provide feedback to the platform. This feedback is crucial to identify which trends are gaining traction and should be integrated natively into the platform.

Naturally, as often in software engineering, this is a trade-off. This section should not be misinterpreted as advocating for building a platform that tries to be everything for everyone. Restricting the platform’s capabilities and providing sensible defaults - which will always be an opinionated decision - is the central idea for reducing complexity. Escape hatches are an elegant way to allow deviations without bloating the platform itself. But teams that constantly need to use those escape hatches - or are only cooking with their own ingredients - are likely better served by not using the platform and using other tools directly.

Templates and Golden Path

While the selection of ingredients - the platform’s capabilities - is already the first step in reducing complexity, it can still be an overwhelming offering for teams. The provided tools and services are often complex and might not be needed in every use case.

To reduce the entry barrier further, it is helpful to guide teams to make a sensible selection for getting started and how to use them effectively. In addition to the documentation, a common approach is to provide project templates and reference implementations for common patterns.

In particular, the templates should form *walking skeletons* (minimal, end-to-end implementations) for common application patterns and should be designed to get teams up and running quickly with minimal setup. They should cover relevant steps of the application lifecycle - from development to evaluation, deployment and monitoring. These templates are often referenced as the *golden path* (or *paved road*) as they define a standardised and opinionated way for teams to build their applications on the platform. Beyond demonstrating the platform’s capabilities, they can also include recommended technology choices at the implementation level. Typically, they include a default project structure with package managers, linting and testing frameworks, CI/CD workflows, and more. In the context of AI applications, it is a good idea to also include an agent framework, a connection to the platform’s LLM provider, as well as sensible defaults for prompt management, evaluation, and monitoring.

Beyond getting teams started quickly, those templates are also a key tool for rapid prototyping and experimentation, which is, as discussed in Chapter 1 crucial in the context of AI applications.

Importantly, as outlined in the previous section, the technology recommendations should not be forced, but rather be encouraged. Knowing that the technology choices are supported by the platform team and used by other teams gives teams confidence that they might not be alone when facing challenges and can be sufficient motivation to follow the golden path. Additionally, the templates are also a great opportunity for directly integrating organisation-wide macro-architecture rules (e.g., logging standards, authentication and authorisation mechanisms, and monitoring approaches) and compliance requirements (e.g., security scanning, licence checking), ultimately helping teams to adhere to organisational standards without having to worry about them.

This already shows that those templates are not trivial and capture many decisions. Therefore, it is important to not overload them and keep them focused on kicking off the most common use cases. They should not be an extensive demonstration of all platform capabilities. More advanced capabilities (in the context of AI, probably fine-tuning and multi-agent orchestration capabilities) should be covered either in separate templates, dedicated reference implementations, or how-to guides.

To understand the rationale behind choosing such a path, another culinary metaphor can be used: *Omakase* (visualised in Figure 2-3).

A concept which originates from Japanese cuisine and translates to *I'll leave it up to you* (or more loosely to *'the chef's choice'*). Imagine you are in a Japanese restaurant and, as you are not familiar with the cuisine, you don't know (yet) what *good* is. So you leave it up to the chef to prepare a meal for you. The chef selects the best ingredients of the day and prepares a meal based on their expertise.

Figure 2-3. Omakase principle

With technology, it's often not that different. The ecosystem is huge and constantly evolving. For newcomers, it's hard to know what *good* is. Similar to the restaurant scenario, you can leave it up to the platform team - the *chefs* - to select the best ingredients and combine them into a delicious meal.

This principle, for instance, served as the foundation for Ruby on Rails.[16]

Naturally, over time, teams will gain more experience and develop their own *taste* for perhaps more advanced ingredients, something the chefs didn't include in the initial meal. This is where the earlier discussed *mise-en-place* principle becomes valuable. They can just pick and extend their use case based on the other ingredients, readily prepared by the chefs.

Developer ~~Experience~~ Joy

Lastly, in the context of DevEx, the aspect of joy should not be overlooked. Research clearly indicates that happy developers are more productive and more likely to stay with the company.[17]

So far, the sections have focused on speeding up delivery of the stream-aligned teams by removing friction. Few things are more frustrating than being slowed down by tools and processes. However, the removal of pain is not the same as the addition of joy.

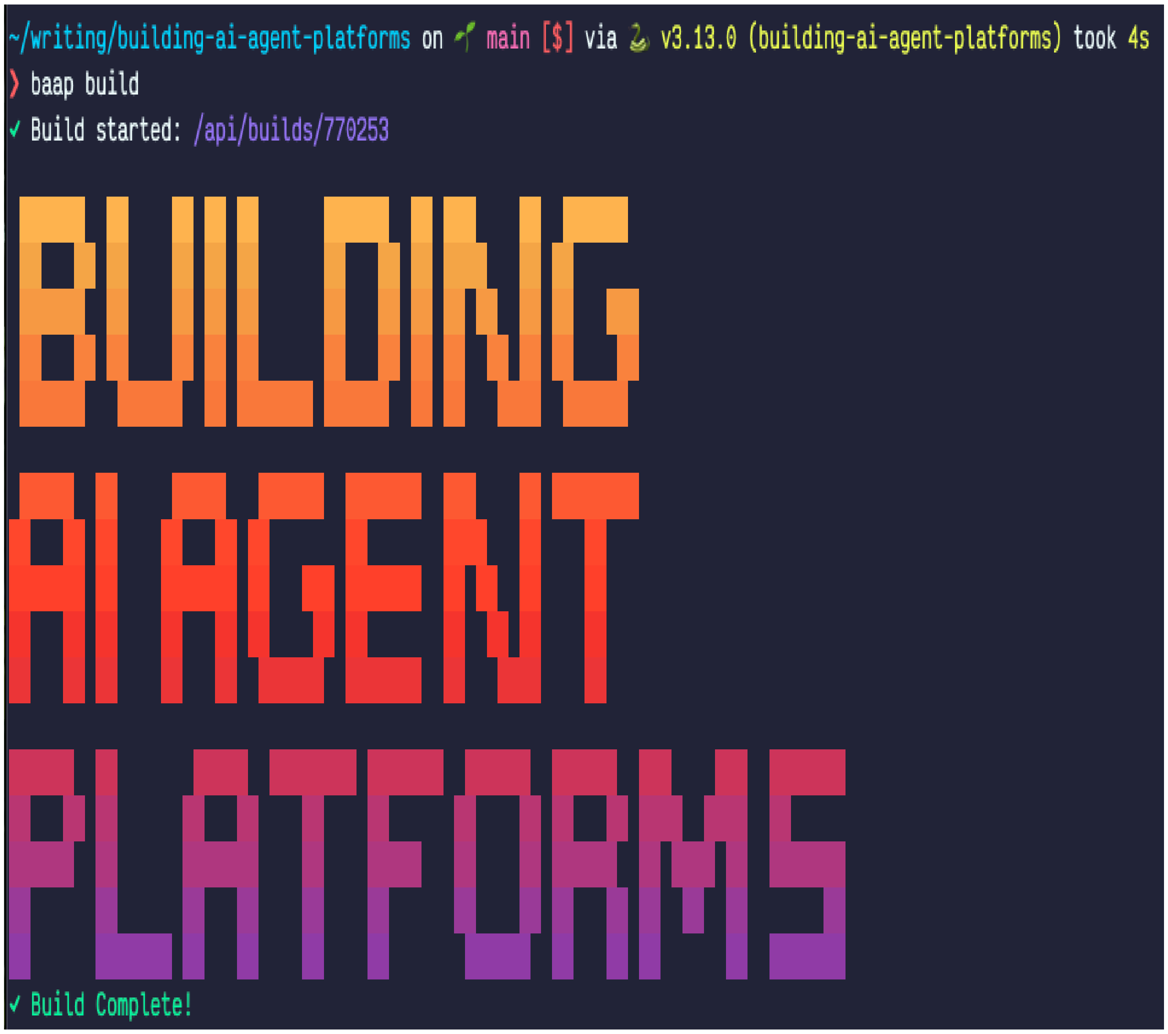
This speaks to a fundamental truth of software development, one that is even captured as the first point in the Agile Manifesto: *Individuals and* *interactions over processes and tools*.[18]

This focus on the *individual* is the key. Aspects that contribute to developer joy include the wish for working with modern tools, having fast feedback loops, and even moments of pure whimsy.

Rapid feedback loops are an essential aspect of developer productivity and, in a way, the ultimate representation of removing friction. This begins with having swift local development setups allowing developers to iterate quickly on their code, and continues with fast deployment pipelines and timely feedback from monitoring and alerting systems from the real production systems.

The wish for *modern tools* is also a source of joy. This encompasses the profound satisfaction of using a tool that feels *magical*, like `k9s`[19] making complex Kubernetes operations feel simple, or `thefuck`[20] intuitively correcting a typo. These tools are joyful because they are thoughtfully crafted to create a state of flow.

Finally, there is the joy of pure delight. This is not about productivity, but about positive reinforcement. For example, using a library like TerminalTextEffects[21] to make a *deploy successful* message animate across the screen. It's not necessary, but it transforms a simple status update into a moment of celebration (Figure 2-4).



```
~/writing/building-ai-agent-platforms on 🌱 main [$] via 🐍 v3.13.0 (building-ai-agent-platforms) took 4s
❯ baap build
✓ Build started: /api/builds/770253

BUILDING
AI AGENT
PLATFORMS
✓ Build Complete!
```

Figure 2-4. How the authors get joy from every new book build!

This is why Developer Joy is a higher bar than good DevEx. Developer Joy is the active addition of satisfaction, whimsy, and empowerment. It's the difference between a tool that is merely usable and a tool that is loveable. It's the ultimate sign that the individuals and their interactions have truly been prioritised over the mere processes and tools.

Cross-functional Requirements

For a platform to be valuable to stream-aligned teams, it must adhere to the cross-functional requirements of those teams. Without diving deep into this area, typical aspects to consider are performance, reliability, and availability of the platform services. But also security and compliance requirements.

Naturally, all data stored by the platform must be handled securely and in compliance with organisational policies (e.g., data privacy). Again, the *platform as a product* mindset is crucial here. The platform team is responsible for running the platform services in production. This includes monitoring, alerting, incident management, and postmortems on the platform.

Platform teams should define, measure, and communicate Service Level Agreements (SLAs) and performance characteristics of their services. This helps to set the right expectations and is essential for stream-aligned teams to make informed decisions when building their applications on top of the platform. Naturally, as with any other product, those SLAs and performance characteristics should be optimised based on the users' needs to ensure the platform has an attractive offering.

A particular challenge for platforms is the so-called *noisy neighbour* problem. As multiple teams will share the same platform services, it is possible that one team consumes a disproportionate amount of resources, negatively impacting the performance and reliability for other teams. It's the platform team's responsibility to monitor and mitigate such issues, for example, by implementing rate limits, resource quotas, or prioritisation mechanisms.

Simplifying Compliance

Having a platform on which all applications are running might give decision makers the false idea of using the platform as a tool for enforcing compliance throughout the organisation. However, this quickly might go against the platform's primary goal of speeding up delivery. If a platform starts gatekeeping and only allows compliant applications to be deployed, it quickly becomes a bottleneck. In reality, compliance is often a complex topic, with not each requirement being applicable to each application. The platform team is unlikely to have the right context and knowledge to make those decisions for individual applications. Especially, a self-service that requires the platform team to review and approve each application before deployment, is not a real self-service anymore and will quickly lead to frustration. Many teams won't see the difference between a platform and central compliance enforcement, and finally will blame the platform for slowing them down, which ultimately hinders adoption.

Naturally, this does not mean that compliance is not an important aspect for platforms. But it needs to be approached smarter. First and foremost, there is a shared responsibility model between the platform team and the stream-aligned teams, similar to public cloud providers and their customers.[22] As discussed in the previous section, the platform team is responsible for ensuring that the platform itself is compliant with organizational policies. The platform needs to document and communicate which compliance requirements are covered by the platform. The stream-aligned teams are responsible for ensuring that their applications built on top of the platform are compliant, and that they use the platform in a compliant way. They are the experts for their applications and have the necessary context (e.g. data classification requirements) to make those decisions, ultimately holding them accountable for the end-to-end compliance of their applications. There should be independent processes and audits that ensure the compliance of applications.

This naturally also includes the previously discussed escape hatches - if teams integrate tools or services outside the platform's offering, they need to guarantee that those are compliant as well. Allowing escape hatches sometimes makes platform teams uncomfortable as it feels like *giving away control*, but in reality it's the only way to make sure team autonomy. Therefore, platform teams should avoid using compliance as an excuse for not allowing deviation.

Besides defining this shared responsibility model, the platform can do more to help teams be compliant. The goal is to make it easy for teams to be compliant. No team will complain if they adhere to compliance requirements automatically. This means the platform should provide sensible defaults that are compliant out-of-the-box, for both the platform services and the provided templates. For example, the platform could provide default logging and observability setups that adhere to data privacy requirements and make it easy to fulfill audit requirements.

Fostering Enablement and Community

Finally, beyond the technical aspects of the platform, it’s crucial to invest in organizational aspects as well.

It is difficult for stream-aligned teams to keep up with the fast-evolving AI ecosystem and the platform’s capabilities. Team Topologies suggests having a dedicated Enabling Team to bridge this gap.[23] This team is not directly responsible for building or maintaining the platform, but focuses on helping stream-aligned teams effectively use the platform and upskill them in AI Engineering practices. Good examples of such activities are training sessions, workshops, and most effectively doing pair programming sessions to be really involved in the stream-aligned team’s workflows. Enabling teams often also contribute to the platform’s documentation, templates, and reference implementations.

Besides that, it is also an established practice to form Communities of Practice (CoP). These target interested individuals and should set the right environment for informal knowledge sharing and learning across teams. Communities can be supported by the platform or enabling team by organizing regular meetups, maintaining discussion forums, and sharing best practices and success stories.

You will know that you are doing something right when you see people contributing back to the platform. This is often referred to as internal open source and is a great way to encourage collaboration and knowledge sharing. This is the pinnacle of the platform’s success. Not only do users benefit from the platform but they see the value so much that they contribute back to the platform for the benefit of all. Whenever you see this make sure that you engage quickly and encourage the contributor. These are great opportunities to shout out to the organisation and thank the contributor for their contribution.

The Additional Benefits of a Successful

Platform

As the earlier discussed definition shows, the main goal of a platform is to accelerate the development and delivery of applications. This is without doubt the primary value proposition of a platform, and alone justifies the investment in building and maintaining a platform for large enough organizations.

However, there are additional advantages and values that a platform can help achieving, which will be outlined in the following sections.

Reducing Operational Overhead

Reducing the cognitive load is not limited to the pure development and delivery of applications. This principle can also be extended to the operational side of applications.

The platform contributes to reducing the operational overhead in two ways. Firstly, providing tools and services (e.g., logging, tracing, alerting) makes it easier for teams to operate their own applications. Secondly, operating central services (e.g., orchestration tools, LLM providers) centrally by the platform team can further reduce the operational burden on individual teams. A prerequisite for this is that the SLOs and performance of those central services meet the needs of the teams, as discussed earlier.

Driving Innovation

As outlined in Chapter 1 in the context of AI applications, experimentation and rapid prototyping are crucial for exploring potential business opportunities. Platforms can play a key role in enabling and accelerating those activities by providing easy on-ramps and sensible defaults for rapid prototyping (e.g. through the earlier outlined templates). Those early experiments likely have fewer non-functional requirements (e.g., load capacity, availability), which should be reflected in the provided templates. Those experiments should not be slowed down by complex configurations and setups that are not needed in the early stages. Nevertheless, having those early experiments already on the platform is a huge advantage, as they can evolve into more mature applications by gradually incorporating more of the platform's offering.

Organisational Upskilling and Enablement

AI Engineering is an exciting and strategically important field for many organisations. It is very common that organisations want to enable and upskill the engineering teams within their organisation. Platforms can contribute to this goal in multiple ways.

By integrating into already existing platforms (see section "Existing Platform Ecosystem"), the AI platform can expose AI capabilities to a broader audience of engineers and allow for a smooth transition when teams want to integrate AI capabilities into their applications. Additionally, by providing easy on-ramps, sensible defaults, and good documentation, the platform can lower the barrier for engineers new to the field of AI and demonstrate good practices. Offering tools and services required for more advanced practices (e.g., Evaluation Management System) simplifies it for teams to adopt those practices.

Finally, by having enablement teams and building a community around the platform, the platform team can facilitate knowledge sharing and learning across the organization's teams.

Driving Governance and Security Standards

Throughout the AI application lifecycle, governance and security are crucial aspects that stream-aligned teams need to address, often defined by central organisational policies and standards.

As with other aspects, this leads to high cognitive load. Thus, many teams delay implementing those requirements until the very last moment when it is enforced by the quality process. Platforms can help by providing compliant services, sensible defaults, and templates that directly integrate compliance-relevant tools. However, as discussed in the success factors section "Simplifying Compliance", it's essential to avoid creating bottlenecks by enforcing compliance as a prerequisite for using the platform. Following the DevOps model *You build it, you run it*, teams know about the context of their applications and are therefore responsible for complying with the organisational rules.

Figure 2-5. XKCD: AI Hiring Algorithm - https://xkcd.com/2237

Existing Platform Ecosystem

As the previous chapter has shown, AI Engineering has a significant overlap with traditional software engineering practices. Therefore, it’s not surprising, that many of the existing platforms in the organisation are likely to already provide capabilities that are relevant for AI applications, as well. To avoid building parallel and potentially competing platforms, it is crucial to evaluate the existing platform ecosystem and identify which capabilities can be reused or extended for AI applications. The following sections outline the most common platform types and their core capabilities and discuss how AI platforms can benefit from and fit into this already existing ecosystem.

Internal Development Platform

The most common platform in an organisation is the Internal Development Platform (IDP). It is targeted at Software Engineers and enables them to build and run software applications (for simplicity think microservices).

IDPs tend to be the largest and best funded platforms in the organisation, and are therefore the most characteristic example of a platform. This is what most people think of when they think of platforms. If the IDP is mature and well-designed, it implements many of the previously outlined universal success factors, while targeted at a different audience.

Often they are built on top of cloud providers and provide a runtime for applications (e.g., Kubernetes clusters) as well as CI/CD systems, logging and monitoring setups, authentication and authorisation mechanisms, and more.[24] Considering the similarities between AI applications and traditional software, those capabilities *could* form a potential foundation for an AI Platform.

Although, their sensible defaults and templates are not necessarily tailored to AI applications. But before building similar capabilities again from scratch, it is likely a better approach to evaluate the existing IDP and potentially extended them and define new sensible defaults for AI applications.

Data Platforms

Data platforms are the next most common platform in the organisation. They often originated from an organisation's goal to become more data-driven and to derive insights from their data assets. Those platforms target data analysts, engineers, and scientists. Typically, they provide capabilities for managing and processing large volumes of data. This often includes orchestrators for data ingestion pipelines, data storage solutions, and data processing frameworks (e.g., ETL/ELT tools, big data processing engines), but also tools for data governance, data quality, and data security.

Considering that data organisation and workflow automation are common AI use cases, it becomes clear that those capabilities might be beneficial for certain AI applications.

Data Mesh Platforms

After the initial wave of data platforms, the large data lakes became monolithic and hard to manage. Often those data lakes turned into data swamps, with low data quality and little trust in the data. To address those challenges, many organisations started adopting the data mesh paradigm, which decentralised the data management and advocates for maintaining several smaller data products instead of one central storage.[25]

Data Mesh Platforms are often an extension to the data platforms and provide capabilities for managing those data products in a decentralised way, e.g. through data product registries, data catalogues, and data discovery tools.

Here too, it goes without saying that those capabilities can be beneficial for AI applications, as AI applications can be both consumers and producers of data products.

MLOps Platforms

While Data Platforms focus on managing data, MLOps Platforms target data scientists and ML engineers. They are designed to support the whole (traditional) MLOps lifecycle, from data preparation, over model training and evaluation to model deployment, all optimised for scale. Depending on the maturity of the platform, this includes capabilities for distributed training, feature stores, experiment tracking, model registries, but also monitoring solutions for detecting data and model drift.

Given that LLMs and other AI models are at the core of AI applications, those capabilities can also be highly relevant for AI applications.

Comparison with Low Code/No Code Platforms

There are many Low Code/No Code AI platforms available, and certainly there are more to come. Many of these are targeted to specific use cases and are not general-purpose platforms. However, the main difference between these platforms and the platforms discussed so far is that they are almost explicitly golden cages. They only provide a limited set of capabilities and are not open to the full customisation available with code. While this can be very enabling for organisations lacking coding expertise, it is no substitute for the customisation available on more general-purpose platforms.

Given how effective AI is at assisting with coding, there has never been a better time for a greater number of people to use code for their work. It is worth bearing in mind that you may well have more and more people using code to solve their issues and make sure the platform is intuitive for inexperienced users. Having an onboarding process that is as simple as possible is a key factor in ensuring that the platform is accessible to everyone. This can include literally teaching employees how to code. You will be surprised at how many people are willing to learn how to code, and how quickly they can pick up the skills needed to be productive.

Integrating Agent Platforms into the Existing Ecosystem

Before diving into the specific capabilities of an AI Agent Platform in the next chapter, it is already clear that many of the existing platforms might provide capabilities that are also relevant for AI applications. Naturally, within an organisation, those platforms are not isolated silos, but rather an integrated ecosystem.

Considering that AI applications themselves are not running in isolation but are often integrated with other applications and services, it also makes sense from a user perspective to use the same tools wherever possible. An example is a centralised logging solution; teams likely want to have all their logs in one place, instead of having a separate system for AI applications.

Additionally, it helps teams who want to integrate AI capabilities into their existing applications (e.g., an agent endpoint in a microservice). They don’t have to learn a completely new platform, but can use the offering they are already familiar with and extend it with capabilities provided by the AI platform.

Another benefit of leveraging the already existing capabilities is that they are already battle-tested. With many applications, data products, and ML models running in production, they most likely satisfy the required performance and availability needs.

Considering all this, rebuilding existing functionalities is a massive undertaking and should be avoided wherever possible. It is often more effective to build on top of existing platforms, and bundle and extend their capabilities together, optimised for AI applications and agents (see Figure 2-6).

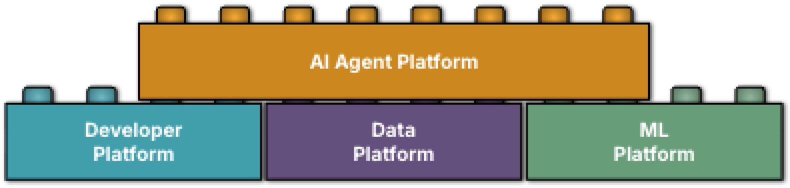


Figure 2-6. AI Agent Platform fitting in existing platform ecosystem

Nevertheless, existing platforms also have their limitations and shortcomings. They have grown over time and often include tech debt that hinders adoption and slows down delivery. Looking at the previously outlined success factors, it is far from trivial to fulfil all of them. Therefore, it’s crucial to prioritise capabilities unique to AI applications and agents and validate them against the existing ones. AI applications might have a new strategic relevance for the organisation, and existing capabilities might not be sufficient to satisfy the new needs. In this case, a strategy for gradually replacing existing platform capabilities might be required.

Conclusion

The hardest thing about building a platform is resisting the urge to build everything at once. The temptation to create the perfect, comprehensive solution is strong, especially when faced with the breadth of capabilities that AI applications might need. But platforms that succeed are the ones that start small, win genuine adoption through value, and grow organically based on real user needs.

Remember that your first platform users should surprise you. If they don’t, you’ve probably built something too restrictive. The goal is not to predict every possible use case, but to provide enough capability and flexibility that teams can solve problems you didn’t anticipate. This means accepting that some teams will bring their own ingredients, use escape hatches, and occasionally do things in ways that make you uncomfortable. That’s not a failure of the platform, it’s evidence of team autonomy.

The AI Engineering landscape will continue to evolve rapidly. Tools that are essential today may be obsolete in six months, which is roughly the same timeline as JavaScript frameworks but with more hype and venture capital. Don’t let this drive you into analysis paralysis or toward building heavyweight abstractions. Focus instead on providing clear on-ramps to well-chosen tools, knowing that you’ll need to adapt as the ecosystem matures. The platform teams who thrive are those who embrace this uncertainty and build with reversible decisions in mind.

Finally, if you’re building an AI platform in an organization that already has IDPs, data platforms, or MLOps capabilities, your first instinct should be to extend rather than replace. The value of a unified ecosystem where teams use consistent tools across all their applications far outweighs the appeal of a greenfield build. Yes, those existing platforms have accumulated debt and constraints. Yes, they have that one service that everyone complains about but no one has time to fix (you are probably lucky if it’s just one service). Yes, there’s probably some configuration file from 2019 that no one understands but everyone is afraid to touch. Build on those foundations anyway. Your AI platform will benefit from years of organizational learning, battle-tested reliability, and most importantly, existing users who already trust the broader platform ecosystem. Build on those foundations, and your AI platform will benefit from years of learning that came before.

The next chapter moves from these organizational and product considerations to the specific technical capabilities that make an AI Agent

Platform effective for building AI applications.

---

1 Team Topologies: Organizing Business and Technology Teams for Fast Flow. *https://www.teamtopologies.com/book*

2 Effective Platform Engineering. *https://www.manning.com/books/effective-platform-engineering*

3 Evan Bottcher, "What I Talk About When I Talk About Platforms," MartinFowler.com, 2018, *https://martinfowler.com/articles/talk-about-platforms.html*

4 What is an Internal Developer Platform (IDP)?. *https://internaldeveloperplatform.org/what-is-an-internal-developer-platform/*

5 Applying product management to internal platforms - Thoughtworks. *https://www.thoughtworks.com/radar/techniques/applying-product-management-to-internal-platforms*

6 XY Problem. *https://xyproblem.info/*

7 Platform Engineering: Innovation Through Harmonization by Gregor Hohpe. *https://architectelevator.com/book/platformstrategy*

8 Wikipedia: Extreme Users. *https://en.wikipedia.org/wiki/Extreme_users*

9 *Invent and Wander*, Jeff Bezos, Walter Isaacson, Harvard Business Review Press. *https://store.hbr.org/product/invent-and-wander-the-collected-writings-of-jeff-bezos-with-an-introduction-by-walter-isaacson/10466*?

10 Platform Engineering: Innovation Through Harmonization by Gregor Hohpe. *https://architectelevator.com/book/platformstrategy*

11 The Diátaxis Framework. *https://diataxis.fr/*

12 Eric Holscher, Write the Docs, Docs as Code. *https://www.writethedocs.org/guide/docs-as-code/*

13 Docs as tests. *https://www.docsastests.com/*

14 API Design by Carson Gross Big Sky Dev Con 2025. *https://www.youtube.com/watch?v=dTstnhS3moc*

15 Platform Engineering: Innovation Through Harmonization by Gregor Hohpe. *https://architectelevator.com/book/platformstrategy*

16 Ruby on Rails - Doctrine, *https://rubyonrails.org/doctrine#omakase*

17 McKinsey: Why your IT organisation should prioritise developer experience. *https://www.mckinsey.com/capabilities/tech-and-ai/our-insights/tech-forward/why-your-it-organization-should-prioritize-developer-experience*

18 Agile Manifesto: *https://agilemanifesto.org/*

19 K9s. *https://k9scli.io/*

20 Github The Fuck. *https://github.com/nvbn/thefuck*

21 TerminalTextEffects. *https://chrisbuilds.github.io/terminaltexteffects/*

22 AWS Shared Responsibility Model. *https://aws.amazon.com/compliance/shared-responsibility-model/*

23 Team Topologies: Organizing Business and Technology Teams for Fast Flow. *https://www.teamtopologies.com/book*

24 Core Components of an Internal Developer Platform (IDP). *https://internaldeveloperplatform.org/core-components/*

25 Data Mesh Principles and Logical Architecture by Zhamak Dehghani. *https://martinfowler.com/articles/data-mesh-principles.html*

About the Authors

Ben O’Mahony is Principal AI Engineer at Thoughtworks. He is a results-driven AI/Engineering leader with a track record of building high-performing teams and shipping business-critical AI, ML and data products and platforms at scale. He has deep expertise across the full Engineering and Data lifecycle from research to production deployment. Ben is adept at defining technical strategy, driving execution and partnering cross-functionally to deliver measurable impact. Recently Ben has been intensely focused on building Generative AI platforms, models and agents.

Fabian Nonnenmacher is a Lead Engineer at Thoughtworks, bringing a unique perspective to AI Engineering. Starting his career as a backend software developer, he also took on a Product Business Analyst role, giving him a distinct skill set that combines robust software architecture with a keen business and product mindset. This dual focus was central to his recent work, where he led the successful delivery of an MLOps platform. This project not only empowered teams to launch projects but also delivered tangible business value. A key part of this effort involved preparing the platform to unlock the full potential of Generative AI.