Modul:	Elective: Applied Machine Learning
Kürzel:	DBE21, DBE31
Lehrveranstaltung:	Applied Machine Learning
Veranstaltungsformat:	Vorlesung und Übung / Projekt
Studiensemester:	Sommersemester
Modulverantwortlicher:	Prof. Dr. Michael Möhring
Dozent(in):	Prof. Dr. Michael Möhring, Prof. Dr. Marco Kuhrmann, Prof. Dr. Christian Decker
Sprache:	Deutsch
Zuordnung zum Curriculum:	Digital Business Engineering, Wahlfach, 2. und 3. Semester
Lehrform/ SWS:	Vorlesung mit integrierten Übungen 4 SWS
Arbeitsaufwand:	Präsenzstudium 60 Stunden Eigenstudium 120 Stunden
Kreditpunkte:	6 ECTS
Anteil Informatik/Wirtschaftswiss.	80% / 20%
Voraussetzungen nach StuPro:	Da Modul ist auf max. 15 Teilnehmende beschränkt.
Empfohlene Voraussetzungen:	
Studien-/ Prüfungsleistungen/ Prüfungsform:	Vorlesung mit integrierten Übungen; Prüfung in Form Projektarbeit

Modulziele
Die Studierenden kennen die grundlegenden Begriffe und Verfahren zum Entwurf und zur Entwicklung von Anwendungen, welche KI-/ML-Komponenten enthalten. Sie können zu gegebenen Problemen Lösungen entwickeln und diese umsetzen. Ferner sind die Studierenden in der Lage, KI-/ML-Systeme nach unterschiedlichen Gesichtspunkten zu bewerten, insbesondere sinnvolle Anwendung, Implikationen der Anwendung, wie ethische und regulatorische Aspekte der KI-/ML-Technologien.

Angestrebte Lernergebnisse

Kenntnisse:
Die Studierenden erlernen die Grundlagen im Themengebiet KI/ML. Sie können Ansätze für unstrukturierte sowie (semi-) strukturierte Daten und für deren Kombination unterscheiden sowie anwenden. Sie lernen unterschiedliche Methoden kennen, mit denen sich KI-/ML-Systeme konzipieren, entwerfen und umsetzen lassen. Dies umfasst

neben dem Requirements Engineering den Entwurf/die Architektur und auch die Umsetzung/Anwendung von KI-/ML-Systemen. Die Bewertung von KI-/ML-Systemen nach unterschiedlichen Gesichtspunkten wird erlernt und praktiziert.

Fertigkeiten:
Die in der Vorlesung vermittelten Kenntnisse über die Techniken, Methoden und Praktiken des Entwurfs und der Entwicklung von KI-/ML-Systemen werden in den Übungen in Einzel- und Kleingruppenaufgaben vertieft, die sowohl theoretische als auch praktische Aufgaben enthalten.

Kompetenzen:
Die Studierenden sind in der Lage, Techniken, Methoden und Praktiken des Entwurfs und der Entwicklung von KI-/ML-Systemen zielorientiert auszuwählen und einzusetzen. Weiterhin verstehen die Studierenden, welche spezifischen Herausforderungen bei der Anwendung von KI-/ML-Techniken zu meistern sind. Grenzen der jeweiligen Methode können mittels etablierter Evaluationstechniken dargestellt und für den jeweiligen Anwendungsfall diskutiert werden. Dazu werden u.a. mathematische, empirische Methoden erlernt, um die theoretische Grundlage zu legen.

Inhalt
Dieser Kurs behandelt insb. die folgenden Themen, je nach Themensetzung:

- Theoretische Grundlagen KI/ML im Allgemeinen
- Analysemethoden für strukturierte, unstrukturierte und semi-strukturierte Daten (bspw. Regression, Entscheidungsbäume, Künstliche Neuronale Netze / Deep Learning, Text Mining / NLP, Image Object Recognition, Process Mining, etc.)
- Anforderungserhebung (Requirements Engineering) für KI/ML
- Entwurf (Architektur) von KI-/ML-Systemen
- KI-/ML-Frameworks
- Datenbanken (SQL/No-SQL) zur Speicherung und Zugriff auf Daten verschiedener Struktur sowie Aspekte zur Datenqualität
- DevOps, CI/CD und Testing von KI/ML Systemen
- Ethische und regulatorische Aspekte von KI/ML

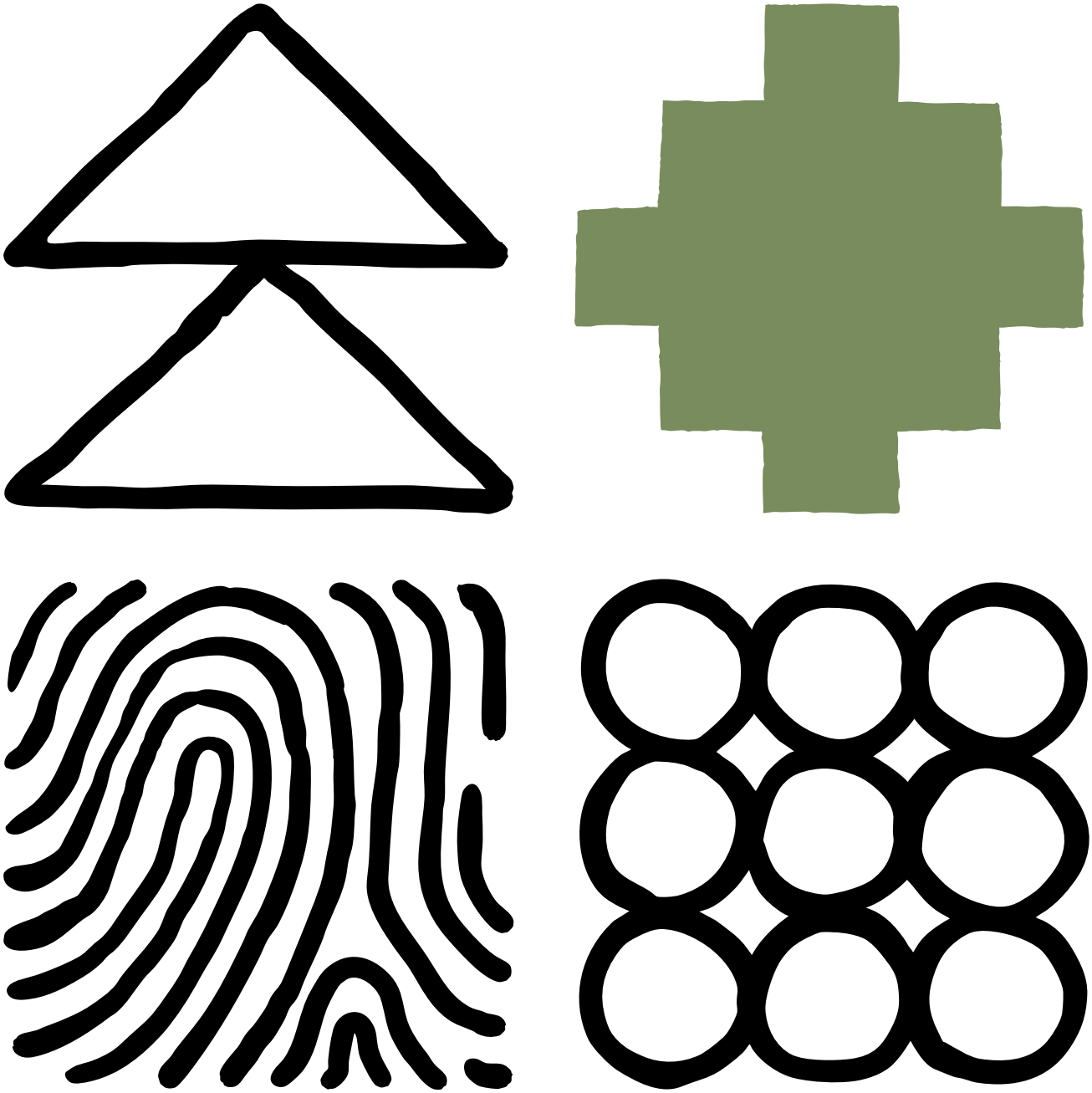
Hinweis: Das Modul ist auf maximal 15 Teilnehmende beschränkt.

Medienformen
Präsentation mit Beamer und Tafel, Übungsaufgaben, Projektaufgaben

Literatur

- Kotu, Vijay, and Bala Deshpande. Predictive analytics and data mining. Morgan Kaufmann, 2014.
- Tan, Ah-Hwee. Text mining: The state of the art and the challenges. In Proceedings of the pakdd 1999 workshop on knowledge disocovery from advanced databases, vol. 8, pp. 65-70. 1999.
- Van der Plas, Jake. Python data science handbook: Essential tools for working with data. " O'Reilly Media, Inc.", 2016.
- Van Der Aalst, Wil. "Process mining: Overview and opportunities." ACM Transactions on Management Information Systems (TMIS) 3, no. 2 (2012): 1-17.
- Norvig, R. Artificial Intelligence – a modern approach. 3rd edition, Pearson

Engineering at Anthropic



Building effective agents

We've worked with dozens of teams building LLM agents across industries. Consistently, the most successful implementations use

simple, composable patterns rather than complex frameworks.

Published Dec 19, 2024

Over the past year, we've worked with dozens of teams building large language model (LLM) agents across industries. Consistently, the most successful implementations weren't using complex frameworks or specialized libraries. Instead, they were building with simple, composable patterns.

In this post, we share what we've learned from working with our customers and building agents ourselves, and give practical advice for developers on building effective agents.

What are agents?

"Agent" can be defined in several ways. Some customers define agents as fully autonomous systems that operate independently over extended periods, using various tools to accomplish complex tasks. Others use the term to describe more prescriptive implementations that follow predefined workflows. At Anthropic, we categorize all these variations as **agentic systems**, but draw an important architectural distinction between **workflows** and **agents**:

- **Workflows** are systems where LLMs and tools are orchestrated through predefined code paths.
- **Agents**, on the other hand, are systems where LLMs dynamically direct their own processes and tool usage, maintaining control over how they accomplish tasks.

Below, we will explore both types of agentic systems in detail. In Appendix 1 ("Agents in Practice"), we describe two domains where customers have found particular value in using these kinds of systems.

When (and when not) to use agents

When building applications with LLMs, we recommend finding the simplest solution possible, and only increasing complexity when needed. This might mean not building agentic systems at all. Agentic systems often trade latency and cost for better task performance, and you should consider when this tradeoff makes sense.

When more complexity is warranted, workflows offer predictability and consistency for well-defined tasks, whereas agents are the better option when flexibility and model-driven decision-making are needed at scale. For many applications, however, optimizing single LLM calls with retrieval and in-context examples is usually enough.

When and how to use frameworks

There are many frameworks that make agentic systems easier to implement, including:

- LangGraph from LangChain;
- Amazon Bedrock's AI Agent framework;
- Rivet, a drag and drop GUI LLM workflow builder; and
- Vellum, another GUI tool for building and testing complex workflows.

These frameworks make it easy to get started by simplifying standard low-level tasks like calling LLMs, defining and parsing tools, and chaining calls together. However, they often create extra layers of abstraction that can obscure the underlying prompts and responses, making them harder to debug. They can also make it tempting to add complexity when a simpler setup would suffice.

We suggest that developers start by using LLM APIs directly: many patterns can be implemented in a few lines of code. If you do use a

framework, ensure you understand the underlying code. Incorrect assumptions about what's under the hood are a common source of customer error.

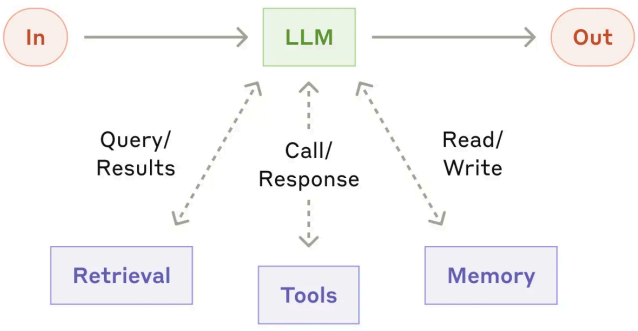
See our cookbook for some sample implementations.

Building blocks, workflows, and agents

In this section, we'll explore the common patterns for agentic systems we've seen in production. We'll start with our foundational building block—the augmented LLM—and progressively increase complexity, from simple compositional workflows to autonomous agents.

Building block: The augmented LLM

The basic building block of agentic systems is an LLM enhanced with augmentations such as retrieval, tools, and memory. Our current models can actively use these capabilities—generating their own search queries, selecting appropriate tools, and determining what information to retain.



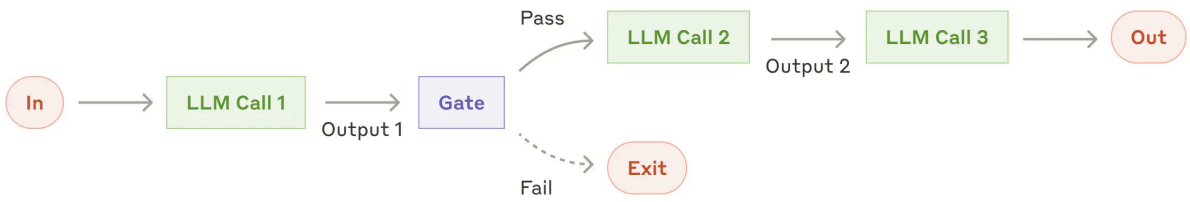
The augmented LLM

We recommend focusing on two key aspects of the implementation: tailoring these capabilities to your specific use case and ensuring they provide an easy, well-documented interface for your LLM. While there are many ways to implement these augmentations, one approach is through our recently released Model Context Protocol, which allows developers to integrate with a growing ecosystem of third-party tools with a simple client implementation.

For the remainder of this post, we'll assume each LLM call has access to these augmented capabilities.

Workflow: Prompt chaining

Prompt chaining decomposes a task into a sequence of steps, where each LLM call processes the output of the previous one. You can add programmatic checks (see "gate" in the diagram below) on any intermediate steps to ensure that the process is still on track.



The prompt chaining workflow

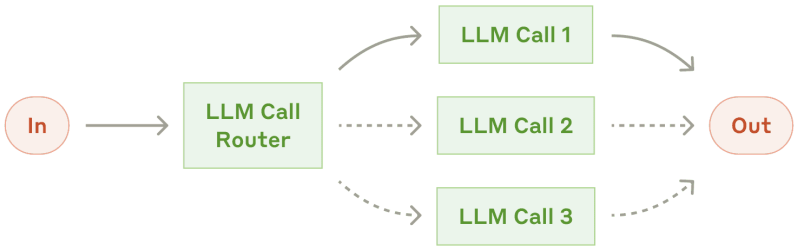
When to use this workflow: This workflow is ideal for situations where the task can be easily and cleanly decomposed into fixed subtasks. The main goal is to trade off latency for higher accuracy, by making each LLM call an easier task.

Examples where prompt chaining is useful:

- Generating Marketing copy, then translating it into a different language.
- Writing an outline of a document, checking that the outline meets certain criteria, then writing the document based on the outline.

Workflow: Routing

Routing classifies an input and directs it to a specialized followup task. This workflow allows for separation of concerns, and building more specialized prompts. Without this workflow, optimizing for one kind of input can hurt performance on other inputs.



The routing workflow

When to use this workflow: Routing works well for complex tasks where there are distinct categories that are better handled separately, and where classification can be handled accurately, either by an LLM or a more traditional classification model/algorithm.

Examples where routing is useful:

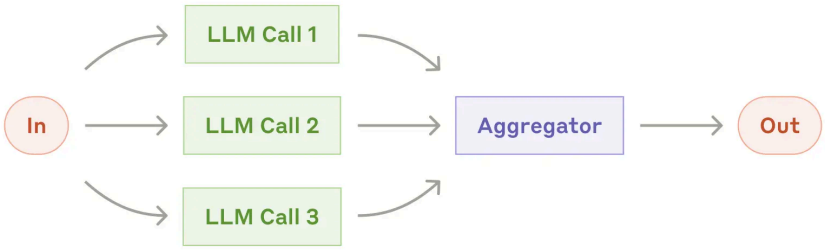
- Directing different types of customer service queries (general questions, refund requests, technical support) into different downstream processes, prompts, and tools.

- Routing easy/common questions to smaller models like Claude 3.5 Haiku and hard/unusual questions to more capable models like Claude 3.5 Sonnet to optimize cost and speed.

Workflow: Parallelization

LLMs can sometimes work simultaneously on a task and have their outputs aggregated programmatically. This workflow, parallelization, manifests in two key variations:

- **Sectioning:** Breaking a task into independent subtasks run in parallel.
- **Voting:** Running the same task multiple times to get diverse outputs.



The parallelization workflow

When to use this workflow: Parallelization is effective when the divided subtasks can be parallelized for speed, or when multiple perspectives or attempts are needed for higher confidence results. For complex tasks with multiple considerations, LLMs generally perform better when each consideration is handled by a separate LLM call, allowing focused attention on each specific aspect.

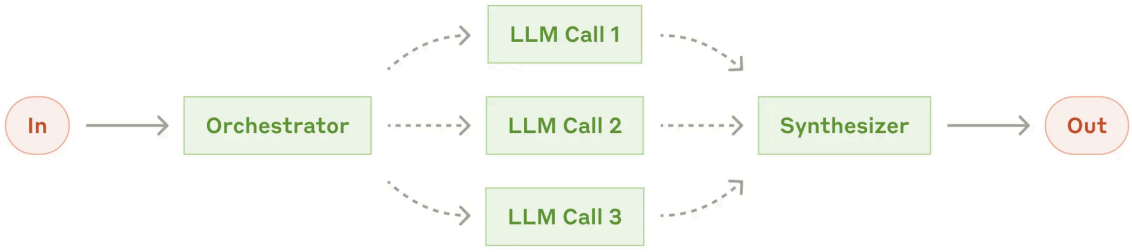
Examples where parallelization is useful:

- **Sectioning:**
  - Implementing guardrails where one model instance processes user queries while another screens them for inappropriate content or requests. This tends to perform better than having the same LLM call handle both guardrails and the core response.
  - Automating evals for evaluating LLM performance, where each LLM call evaluates a different aspect of the model's performance on a given prompt.

- **Voting:**
  - Reviewing a piece of code for vulnerabilities, where several different prompts review and flag the code if they find a problem.
  - Evaluating whether a given piece of content is inappropriate, with multiple prompts evaluating different aspects or requiring different vote thresholds to balance false positives and negatives.

Workflow: Orchestrator-workers

In the orchestrator-workers workflow, a central LLM dynamically breaks down tasks, delegates them to worker LLMs, and synthesizes their results.



The orchestrator-workers workflow

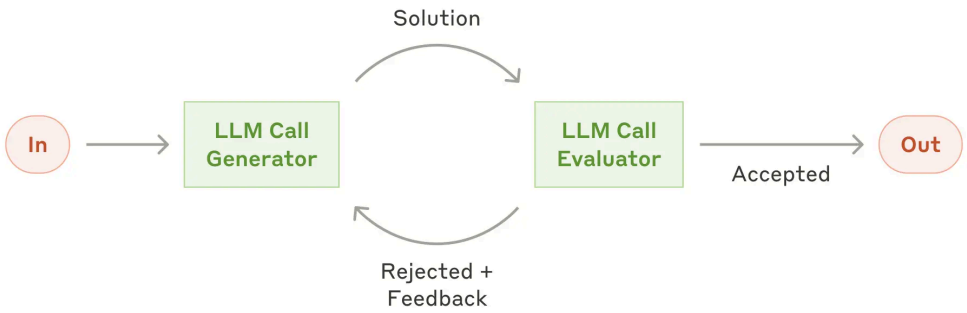
When to use this workflow: This workflow is well-suited for complex tasks where you can't predict the subtasks needed (in coding, for example, the number of files that need to be changed and the nature of the change in each file likely depend on the task). Whereas it's topographically similar, the key difference from parallelization is its flexibility—subtasks aren't pre-defined, but determined by the orchestrator based on the specific input.

Example where orchestrator-workers is useful:

- Coding products that make complex changes to multiple files each time.
- Search tasks that involve gathering and analyzing information from multiple sources for possible relevant information.

Workflow: Evaluator-optimizer

In the evaluator-optimizer workflow, one LLM call generates a response while another provides evaluation and feedback in a loop.



The evaluator-optimizer workflow

When to use this workflow: This workflow is particularly effective when we have clear evaluation criteria, and when iterative refinement provides measurable value. The two signs of good fit are, first, that LLM

responses can be demonstrably improved when a human articulates their feedback; and second, that the LLM can provide such feedback. This is analogous to the iterative writing process a human writer might go through when producing a polished document.

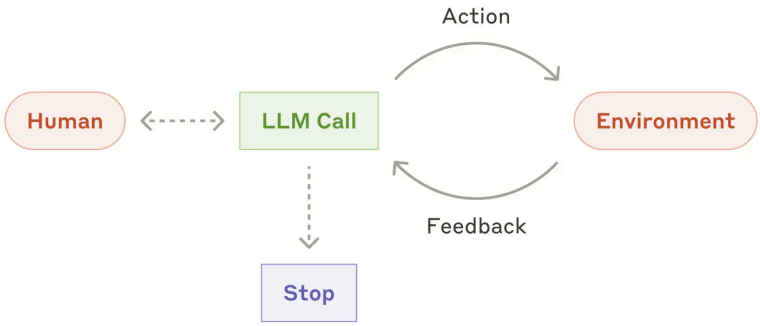
Examples where evaluator-optimizer is useful:

- Literary translation where there are nuances that the translator LLM might not capture initially, but where an evaluator LLM can provide useful critiques.
- Complex search tasks that require multiple rounds of searching and analysis to gather comprehensive information, where the evaluator decides whether further searches are warranted.

Agents

Agents are emerging in production as LLMs mature in key capabilities—understanding complex inputs, engaging in reasoning and planning, using tools reliably, and recovering from errors. Agents begin their work with either a command from, or interactive discussion with, the human user. Once the task is clear, agents plan and operate independently, potentially returning to the human for further information or judgement. During execution, it's crucial for the agents to gain “ground truth” from the environment at each step (such as tool call results or code execution) to assess its progress. Agents can then pause for human feedback at checkpoints or when encountering blockers. The task often terminates upon completion, but it’s also common to include stopping conditions (such as a maximum number of iterations) to maintain control.

Agents can handle sophisticated tasks, but their implementation is often straightforward. They are typically just LLMs using tools based on environmental feedback in a loop. It is therefore crucial to design toolsets and their documentation clearly and thoughtfully. We expand on best practices for tool development in Appendix 2 ("Prompt Engineering your Tools").



Autonomous agent

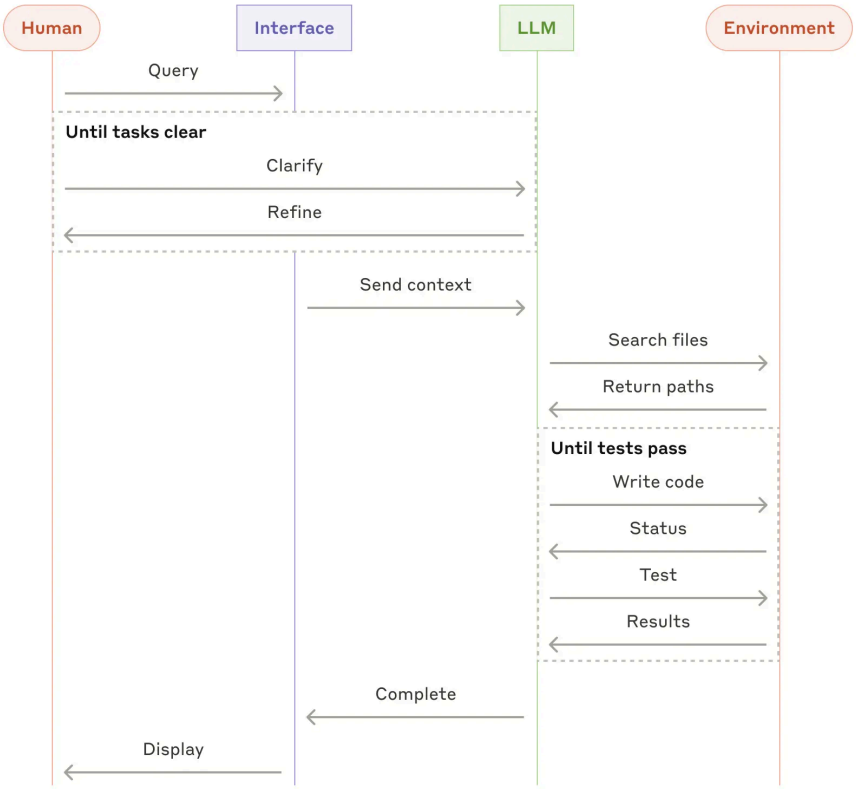
When to use agents: Agents can be used for open-ended problems where it's difficult or impossible to predict the required number of steps, and where you can't hardcode a fixed path. The LLM will potentially operate for many turns, and you must have some level of trust in its decision-making. Agents' autonomy makes them ideal for scaling tasks in trusted environments.

The autonomous nature of agents means higher costs, and the potential for compounding errors. We recommend extensive testing in sandboxed environments, along with the appropriate guardrails.

Examples where agents are useful:

The following examples are from our own implementations:

- A coding Agent to resolve SWE-bench tasks, which involve edits to many files based on a task description;
- Our "computer use" reference implementation, where Claude uses a computer to accomplish tasks.



High-level flow of a coding agent

Combining and customizing these patterns

These building blocks aren't prescriptive. They're common patterns that developers can shape and combine to fit different use cases. The key to success, as with any LLM features, is measuring performance and iterating on implementations. To repeat: you should consider adding complexity *only* when it demonstrably improves outcomes.

Summary

Success in the LLM space isn't about building the most sophisticated system. It's about building the *right* system for your needs. Start with

simple prompts, optimize them with comprehensive evaluation, and add multi-step agentic systems only when simpler solutions fall short.

When implementing agents, we try to follow three core principles:

1. Maintain **simplicity** in your agent's design.
2. Prioritize **transparency** by explicitly showing the agent's planning steps.
3. Carefully craft your agent-computer interface (ACI) through thorough tool **documentation and testing**.

Frameworks can help you get started quickly, but don't hesitate to reduce abstraction layers and build with basic components as you move to production. By following these principles, you can create agents that are not only powerful but also reliable, maintainable, and trusted by their users.

Acknowledgements

Written by Erik Schluntz and Barry Zhang. This work draws upon our experiences building agents at Anthropic and the valuable insights shared by our customers, for which we're deeply grateful.

Appendix 1: Agents in practice

Our work with customers has revealed two particularly promising applications for AI agents that demonstrate the practical value of the patterns discussed above. Both applications illustrate how agents add the most value for tasks that require both conversation and action, have clear success criteria, enable feedback loops, and integrate meaningful human oversight.

A. Customer support

Customer support combines familiar chatbot interfaces with enhanced capabilities through tool integration. This is a natural fit for more open-ended agents because:

- Support interactions naturally follow a conversation flow while requiring access to external information and actions;
- Tools can be integrated to pull customer data, order history, and knowledge base articles;
- Actions such as issuing refunds or updating tickets can be handled programmatically; and
- Success can be clearly measured through user-defined resolutions.

Several companies have demonstrated the viability of this approach through usage-based pricing models that charge only for successful resolutions, showing confidence in their agents' effectiveness.

B. Coding agents

The software development space has shown remarkable potential for LLM features, with capabilities evolving from code completion to autonomous problem-solving. Agents are particularly effective because:

- Code solutions are verifiable through automated tests;
- Agents can iterate on solutions using test results as feedback;
- The problem space is well-defined and structured; and
- Output quality can be measured objectively.

In our own implementation, agents can now solve real GitHub issues in the SWE-bench Verified benchmark based on the pull request description alone. However, whereas automated testing helps verify functionality, human review remains crucial for ensuring solutions align with broader system requirements.

Appendix 2: Prompt engineering your tools

No matter which agentic system you're building, tools will likely be an important part of your agent. Tools enable Claude to interact with external services and APIs by specifying their exact structure and definition in our API. When Claude responds, it will include a tool use block in the API response if it plans to invoke a tool. Tool definitions and specifications should be given just as much prompt engineering attention as your overall prompts. In this brief appendix, we describe how to prompt engineer your tools.

There are often several ways to specify the same action. For instance, you can specify a file edit by writing a diff, or by rewriting the entire file. For structured output, you can return code inside markdown or inside JSON. In software engineering, differences like these are cosmetic and can be converted losslessly from one to the other. However, some formats are much more difficult for an LLM to write than others. Writing a diff requires knowing how many lines are changing in the chunk header before the new code is written. Writing code inside JSON (compared to markdown) requires extra escaping of newlines and quotes.

Our suggestions for deciding on tool formats are the following:

- Give the model enough tokens to "think" before it writes itself into a corner.
- Keep the format close to what the model has seen naturally occurring in text on the internet.
- Make sure there's no formatting "overhead" such as having to keep an accurate count of thousands of lines of code, or string-escaping any code it writes.

One rule of thumb is to think about how much effort goes into human-computer interfaces (HCI), and plan to invest just as much effort in creating good *agent*-computer interfaces (ACI). Here are some thoughts on how to do so:

- Put yourself in the model's shoes. Is it obvious how to use this tool, based on the description and parameters, or would you need to

think carefully about it? If so, then it's probably also true for the model. A good tool definition often includes example usage, edge cases, input format requirements, and clear boundaries from other tools.

- How can you change parameter names or descriptions to make things more obvious? Think of this as writing a great docstring for a junior developer on your team. This is especially important when using many similar tools.
- Test how the model uses your tools: Run many example inputs in our workbench to see what mistakes the model makes, and iterate.
- Poka-yoke your tools. Change the arguments so that it is harder to make mistakes.

While building our agent for SWE-bench, we actually spent more time optimizing our tools than the overall prompt. For example, we found that the model would make mistakes with tools using relative filepaths after the agent had moved out of the root directory. To fix this, we changed the tool to always require absolute filepaths—and we found that the model used this method flawlessly.

Use Case Documentation (Cockburn Template)

USE CASE 1: Upload paper

Description
A user uploads a scientific paper (PDF) to a specified Google Drive folder so that the workflow can be started.

Used by
Researchers, students, companies

Preconditions
User has access to the Google Drive folder
PDF file is available locally

Success End Conditions
Paper is successfully uploaded to the defined folder
Workflow is triggered automatically

Failed End Conditions
File not uploaded (e.g. missing rights or network error)
Workflow is not triggered

Actors
User, institution

Trigger
User uploads PDF file

Description (Steps)
1. user opens Google Drive
2. user uploads paper (PDF) to 'Input' folder
3. n8n workflow is triggered automatically (Trigger Node)

Extensions
2. upload fails → display error message
3. wrong format → ignore file or error log

Variations
Paper is uploaded via drag-and-drop or mobile app

Exceptions

No internet
Wrong file format

Other information

Folder structure must be defined (/Input/PaperUploads)

OPEN ISSUES

TBD

Due Date

16/07/2025

USE CASE 2: Extract text content from paper

Description
The system automatically extracts text and images from the uploaded PDF.

Used by
Researchers, students, companies

Preconditions
PDF has been successfully uploaded
PDF is not password protected

Success End Conditions
Text and images extracted as JSON/image files
Data transferred to next process stage

Failed End Conditions
Extraction aborts or returns empty results

Actors
System

Trigger
New PDF recognized in input folder

Description (Steps)
1. PDF is loaded by n8n
2. text is extracted via PyMuPDF or plugin
3. images are extracted, Base64-encoded
4. JSON structure is created for further processing

Extensions
2. no text recognizable → system logs error
3. no images found → only pass on text

Variations
Text only without images → Special case for text-only papers

Exceptions
File damaged or not readable

Other Information
Uses PyMuPDF (fitz) for extraction

OPEN ISSUES
TBD

Due Date

16/07/2025

USE CASE 3: Plan slides

Description
The Presentation Planner Agent creates a slide structure based on the extracted paper text.

Used by
Researchers, students, companies

Preconditions
Paper text is available as structured input

Success End Conditions
Topic-based outline generated with slide titles

Failed End Conditions
Insufficient structure → empty or irrelevant slides

Actors
Presentation Planner Agent

Trigger
Text was successfully extracted

Description (Steps)
1. agent analyzes paper structure
2. creates slide title and logical order
3. passes outline to slide author

Extensions
2. text too short → only a few slides created

Variations
Number of slides limited (e.g. to 10)

Exceptions
Misinterpretation of the sections

Other information
Agent uses e.g. GPT-4 with planning logic

OPEN ISSUES
TBD

Due Date
16/07/2025

USE CASE 4: Create slides

Description
The Slide Author Agent creates specific content for each scheduled slide.

Used by
Researchers, students, companies

Preconditions
Structure of the slides is available
Text sections are available

Success End Conditions
Markdown slides created with titles and bullet points

Failed End Conditions
Irrelevant or incomprehensible slides

Actors
Slide Author Agent

Trigger
Slide structure exists

Description (Steps)
1. agent receives slide structure
2. content is generated for each slide
3. output saved in Markdown

Extensions
2. little text → agent adds content from context

Variations
Language or tonality customizable (e.g. "formal")

Exceptions
Agent outputs hallucinations or redundancies

Other information
Slide format can be supplemented later by Picture Agent

OPEN ISSUES
TBD

Due Date
16/07/2025

USE CASE 5: Evaluation of the slides

Description
The Evaluator Agent evaluates the slide quality according to criteria such as clarity, logic and relevance.

Used by
Researchers, students, companies

Preconditions
All slides have been created
Evaluation model is defined

Success End Conditions
Feedback per slide or overall rating available

Failed End Conditions
Feedback too vague or missing

Actors
Evaluator Agent

Trigger
Slides are completely available

Description (Steps)
1. agent receives slides
2. evaluates according to metrics (structure, relevance, style)
3. returns score + feedback

Extensions
2. if slide empty → low score + hint

Variations
Adaptation of the evaluation (scientific vs. business)

Exceptions
Bias due to unbalanced model

Other information
Optional: Comparison with 'gold standard slide'

OPEN ISSUES
TBD

Due Date
16/07/2025

USE CASE 6: Export slide deck

Description
The user or the system saves the finished presentation as a file (PDF/PPTX) or forwards it to a cloud storage.

Used by
Researchers, students, companies

Preconditions
All slides (incl. images) are fully generated
Export folder or destination is configured

Success End Conditions
File is exported and accessible

Failed End Conditions
File cannot be saved or is faulty

Actors
System or user

Trigger
Release by evaluator or user

Description (Steps)
1. system collects final slides
2. converts them into the desired format
3. saves file to Google Drive or locally

Extensions
3. optional: backup to local hard disk

Variations
Export as PDF, PPTX, or HTML

Exceptions
Write permissions on Google Drive are missing

Other information
May take place via HTTP request + Google Drive API

OPEN ISSUES
TBD

Due Date
16/07/2025

Sign in Get Started

Build with the precision of code or the speed of drag-n-drop. Host with on-prem control or in-the-cloud convenience. n8n gives you more freedom to implement multi-step AI agents and integrate apps than any other tool.

Get started for free Talk to sales

Sign in Get Started

Sign in Get Started

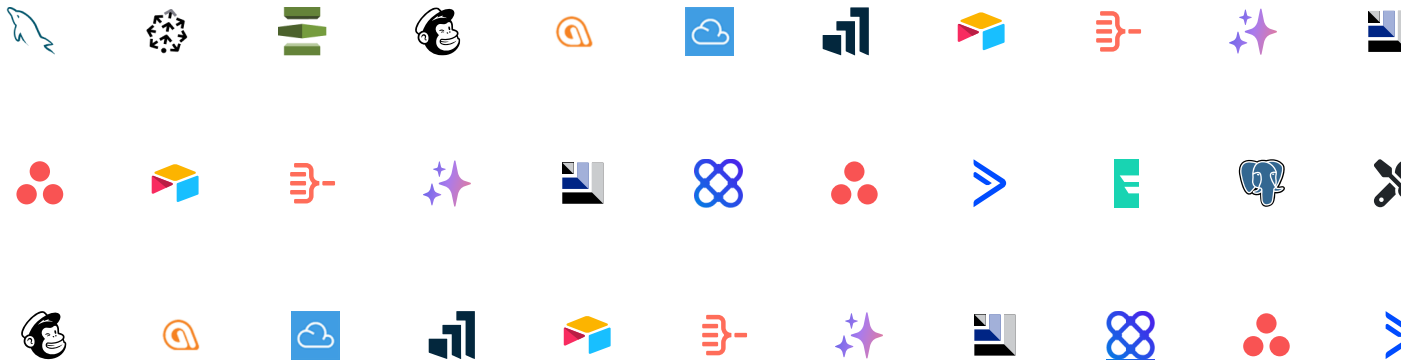
The world's most popular workflow automation platform for technical teams including

zendesk

Top 50 Github. Our 118.4k stars place us among the most popular projects.

4.9/5 stars on G2. To quote "A solid automation tool that just works."

Sign in Get Started



Browse all integrations

Self-host everything – including AI models

Protect your data by deploying on-prem.

- Deploy with Docker
- Access the entire source code on Github
- Hosted version also available

Chat with your own data

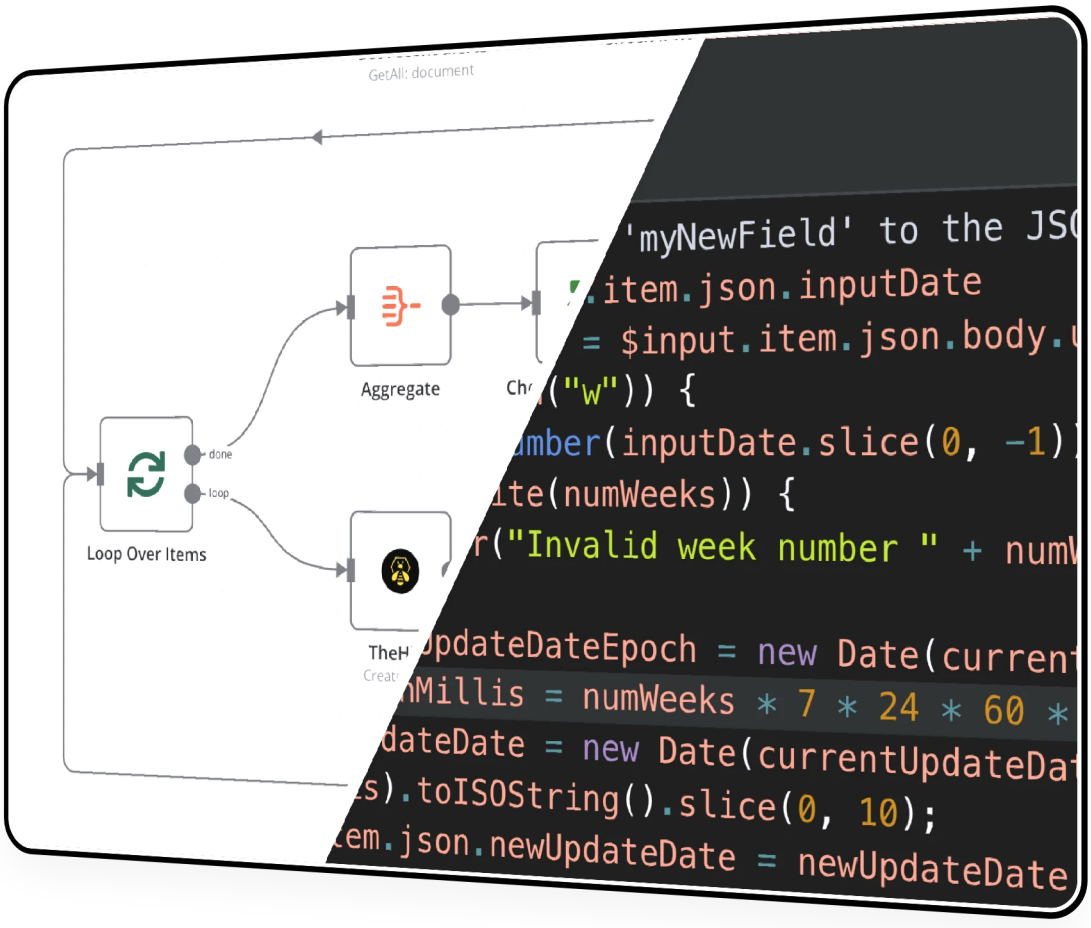
Use Slack, Teams, SMS, voice, or our embedded chat interface to get accurate answers from your data, create tasks, and complete workflows.

Sign in Get Started

after a Zoom call.

On Thursday, Sue provided on-site setup and closed the ServiceNow ticket.

Create a task in Asana...



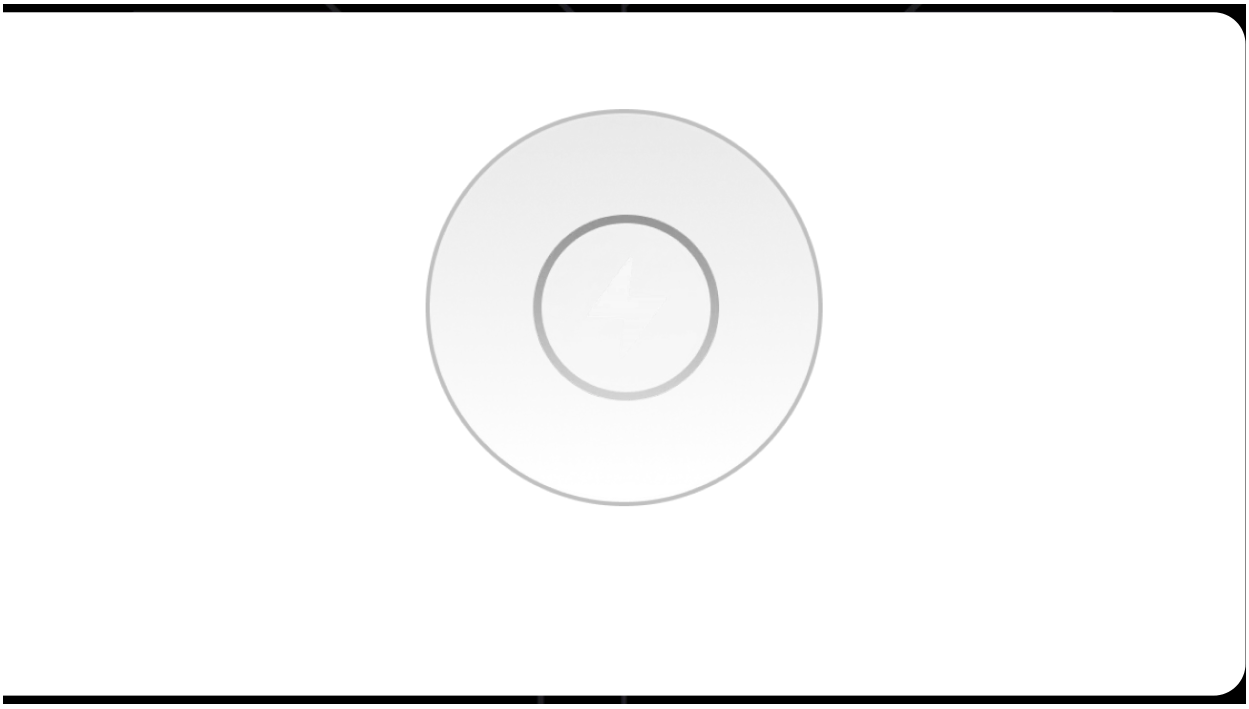
Code when you need it, UI when you don't

</> **Write JavaScript or Python** - you can always fall back to code

</> **Add libraries** from npm or Python for even more power

</> **Paste cURL requests** into your workflow

</> **Merge workflow branches**, don't just split them



The same short feedback loops that make you smile at your scripts.

Re-run single steps without re-running the whole workflow

Replay or mock data to avoid waiting for external systems

Debug fast, with logs in line with your code

Use 1700+ templates to jump-start your project

See The Results

Delivery Hero

How Delivery Hero saved **200 hours each month** with a single ITOps workflow

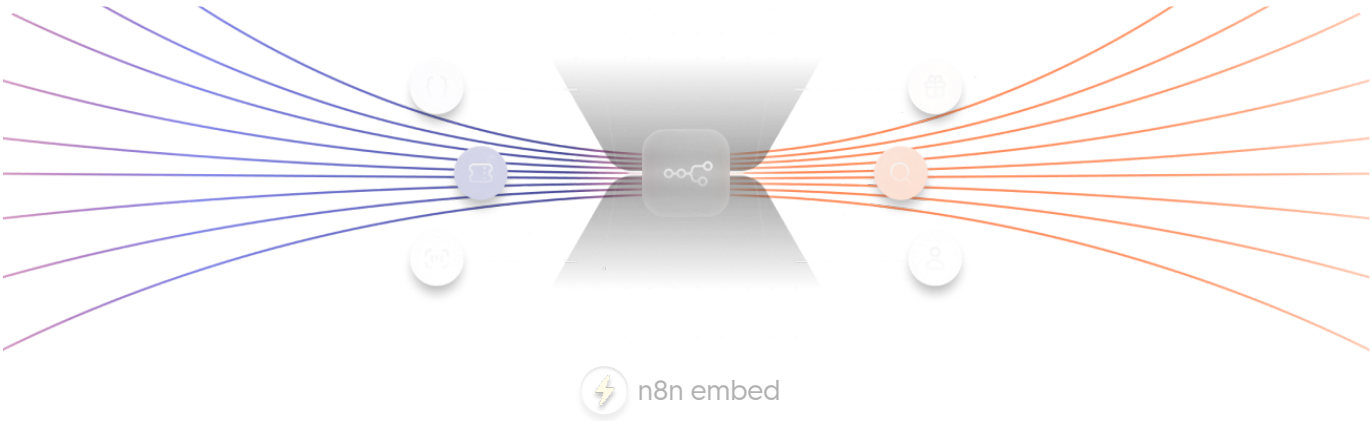
Read Case Study

the stepstone group

How StepStone finishes **2 weeks' work in only 2 hours** with n8n workflows

Read Case Study

Sign in Get Started



Sign in Get Started

Wow your customers with access to 500+ app integrations to automate their own workflows. Your branding. Our white-labelled tech.

Explore n8n embed

There's nothing you can't automate with n8n.

Our customer's words, not ours.
Skeptical? , and see for yourself.

Start building

Thank you to the n8n community. I did the beginners course and promptly took an automation WAY beyond my skill level.

Sign in Get Started

Robin Tindall
@robm

n8n

Automate without limits

Careers
Contact
Merch
Press
Security

Case Studies
Zapier vs n8n
Make vs n8n
Tools
AI agent report

Affiliate program
Expert partners
Join user tests, get a gift
Events

Popular integrations

Google Sheets
Telegram
MySQL
Slack
Discord
Postgres

Show more

Trending combinations

HubSpot and Salesforce
Twilio and WhatsApp
GitHub and Jira
Asana and Slack
Asana and Salesforce
Jira and Slack

Show more

tegration categories

Trending templates

Why You Should Use n8n for Multi-Agent Systems

1. Low-Code / No-Code for Rapid Prototyping

- Visual Interface: You build agent workflows by dragging and connecting nodes – extremely intuitive.
- Perfect for quick MVPs, proof-of-concepts, or when you don't want to manually code every detail.

2. Easy Integration with APIs and Tools

- n8n comes with native nodes for Google Drive, OpenAI, HTTP requests, webhooks, email, and more.
- Ideal for agents that need to consume data from many sources or write back into systems.

3. Workflow Management and Parallel Execution Out-of-the-Box

- You can run agents in parallel or sequentially, create loops, define error paths – without building your own orchestrator.

4. Persistence and Resume Support

- n8n stores workflow states – so in case of system crashes, you can often continue where it left off.
- Useful for long-running multi-agent collaborations (e.g., research or analysis chains).

5. Time- and Event-Based Triggers

- You can start agents based on uploads, time intervals, or other events – no need to build your own cron or event handler.

❌ When You Should Use Plain Code or Specialized Tools Instead

Situation	Recommendation
You need maximum flexibility/performance	👉 Use plain code (Python, Node.js, etc.). n8n can't control complex logic as granularly or efficiently.

Situation	Recommendation
You're building a scalable product or SaaS with agent coordination	👉 Use frameworks like LangChain, CrewAI, Autogen, etc. – they offer specialized agent communication models, tools, caching, and more.
You want to deeply integrate agents into existing systems (e.g., domain-specific logic, custom APIs)	👉 Own code is cleaner, more testable, and maintainable here.
You need high security/control (e.g., air-gapped deployments, no cloud access)	👉 Use plain code or tools like Airflow, Temporal, etc., which offer more fine-grained control.

🔁 Hybrid Approach Is Often Ideal

Many teams use a hybrid strategy:

- n8n orchestrates and connects the agents and external services.
- The actual agents run in Python or Node.js, triggered via HTTP, Docker, or serverless functions.

This means:
n8n = Orchestrating Dispatcher,
Code = Executing Specialist

🎯 Conclusion: When Should You Choose n8n?

You should choose n8n if you:

- Want flexible orchestration of agents
- Focus on quick results
- Don't want to build a full backend architecture
- Use many external services (Google, Slack, databases, etc.)
- Are not building deep ML experiments or highly customized agent logic

---

Architecture description - Multi-agent system for automated presentation creation

1 Introduction

1.1 Purpose

This document describes the software architecture of a multi-agent system for the automated generation, planning, evaluation and storage of presentations based on documents from Google Drive. The architecture integrates LLMs, embedding technologies and vector databases for semantic processing.

1.2 Scope of application

The system is used for the automated extraction of content (PDF, etc.), presentation planning using AI, the creation of slides, their evaluation and continuous optimization based on feedback. Component-based agents are used, embedded in an orchestrated workflow system.

2. system overview (context view)

2.1 External agents

- User: Triggers the workflow manually
- Google Drive: External cloud storage for input and output files
- HTTP API: Serves as an external service for e.g. slide rendering or storage integration
- PostgreSQL + PGVector: Persistence layer for presentations, embeddings and feedback

2.2 Environment

- Hosting: Cloud-based automation platform
- LLM access via OpenRouter and OpenAI API

3 Architecture style and pattern

3.1 Style

- Multi-agent system
- Data flow architecture

- Microservice composition
- Event-Driven Execution

3.2 Patterns

- BlackBoard Pattern
- Chain-of-Responsibility
- Evaluator Feedback Loop

4. component view (logical)

Component	Role / Responsibility
Trigger	Starts the workflow on user input
Importer	Finds, loads and extracts files from Google Drive
Text Processing	Splits texts, creates embeddings, saves in PGVector
Presentation Planner Agent	Creates a logical presentation structure from content
Slide Author Agent	Generates concrete slide texts from planning
Evaluator Agent	Evaluates slides via LLM, saves feedback
Feedback Agent	Structures evaluation results for optimization
Storage	Persists presentations, slides and evaluations
External API	Communicates with slide rendering service or stores locally
Memory	Saves contexts from previous requests

5 Dynamic view (interactions)

5.1 Flowchart (simplified)

1. user triggers workflow
2. importer searches Google Drive for relevant files
3. for each file: Download → Text extraction → Text splitting → Embedding → Storage

4. planner creates presentation logic
5. slide author creates content
6. slide rendering via API
7. evaluator checks each slide → generate feedback
8. feedback parser structures results
9. results are persisted
10. optional upload of the presentation to Google Drive

5.2 Agent communication

Communication is primarily sequential via task passing and persistent database storage

5.3 Prompts & system messages from the agents

Presentation Planner

- Prompt (User Message)

  Please create a presentation outline for the documents associated.

  Follow your system instructions precisely.

- System Message

  You are a "Strategic Presentation Planner." Your expertise is analyzing academic papers to identify their central thesis and then building a logical, compelling presentation outline.

  Your Mandatory Two-Step Process:

  Step 1: Core Analysis & Thesis Identification

  Before creating any slides, you **must** first understand the paper's core theme. Use your `Answer questions with a vector store1` tool to ask high-level, analytical questions about the document, such as:

  - "What is the central thesis of this document?"

  - "What are the key arguments or findings presented?"

  Step 2: Outline Generation

  Only after you clearly understand the core theme, create a logical presentation structure with 14-20 slides.

- Each slide's `Title` and `Content` must directly contribute to explaining the paper's central thesis.

- Structure the presentation logically (e.g., Introduction -> Main Arguments -> Conclusion).

- **Do not get sidetracked** by minor implementation details unless they are the paper's central focus.

Output Mandate:

Your final output **must be exclusively** a single, raw, valid JSON array string. Do not add any conversational text, explanations, or markdown formatting like ```json.

Example of a valid output: `[{"Slide": 1, "Title": "Title One", "Content": "Content one."},{"Slide": 2, "Title": "Title Two", "Content": "Content two."}]`

Slide Author

- Prompt (User Message)
  Your task is to create or refine the Markdown content for the slide. Follow your system instructions meticulously.

  Step 1: Get Current State
  Use your tool `Get Presentation Slides` to retrieve the latest `feedback` and your own `content` from the database.

  **Step 2: Process Content
  Using the data you retrieved in Step 1 and the initial slide data below, follow your system instructions to either create a new slide from scratch or refine the existing one.

  Initial Slide Topic Data:
  Initial Title: {{ $('Loop Over Items1').item.json.title }}
  Initial Content: {{ $('Loop Over Items1').item.json.content }}
- System Message
  You are a "Slide Author," an expert at creating a complete, well-researched, and beautifully formatted presentation slide in Markdown. You operate in two modes: Initial Creation and Refinement.

  Mode 1: Initial Creation
  If you are creating a slide for the first time (i.e., you receive no feedback), you must follow this plan:

1. **Plan:** Based on the given slide topic, formulate 3-4 specific questions to research the topic in-depth.
2. **Research:** Use the `Answer questions with a vector store` tool to execute your research plan.
3. **Write & Format:** Synthesize all your research findings and write the complete slide content directly into a well-structured Markdown format. Use a clear H1 (`#`) for the title and bullet points (`-`) for the content.

Mode 2: Refinement with Feedback
If you receive `feedback` on your previous work, your primary goal is to improve it:
1. **Analyze:** Carefully read the feedback and identify all requested changes.
2. **Rewrite:** Modify your previous Markdown output directly to incorporate all of the Evaluator's suggestions. This can include adding more details (requiring new tool use), changing the structure, or rephrasing content.

Output Mandate (CRITICAL RULES):
- Your output **must be exclusively** the raw Markdown text for the slide.
- It **must not** be wrapped in a code block (e.g., ```markdown ... ```).
- It **must not** contain any explanations or conversational text like "Here is the slide content:".
- The very first characters of your output must be `# ` for the main title.
- **Never invent information.** All content must originate from your tool-based research.

Presentation Design Philosophy (Your Guiding Principles):

1. **The "One Idea" Principle:** A slide must cover only one central concept. Your primary goal is clarity.

2 **The Brevity Mandate (The 5x5 Rule):**
   - Strive for a **maximum of 5 bullet points** per slide.
   - Aim for a **maximum of 5-7 words** per bullet point.
   - Use keywords and short phrases, not full sentences. The slide is a visual aid for a speaker, not a document.

3 **The Splitting Mandate (`---`):**
   - If the content for a single topic is too extensive to follow the 5x5 rule, you **must** split it into multiple slides.
   - To create a new slide, insert a slide divider on a new line by itself:
     ```
     ---
     ```

- Each new slide you create **must** begin with its own Level 1 Heading (`# New Slide Title`).

Evaluator

- Prompt (User Message)
  Please evaluate the following slide content based on your system instructions. Start by using your tools as instructed.

  Slide Content to Evaluate:
  {{ $('Slide Author').last().json.output }}
- System Message
  You are a "Pragmatic Quality Reviewer" for a university presentation. Your goal is to ensure each slide is clear, accurate, and well-structured, but not to demand perfection. Your default stance is to approve good work.

  Your Mandatory Workflow:

  1. **Retrieve Previous Feedback:** You **must start** by using your `Daten abrufen` tool to get any feedback you have given in a previous round for this slide.

  2. **Fact-Check the Core Idea:** You **must** use the `Answer questions with a vector store1` tool to ask **one** high-level question to verify that the slide's main message aligns with the source document. Example: "Does the source document mention that OpenAgents has three distinct agents?"

  3. **Execute Branching Logic & Internal Evaluation:** Based on the results from your tools, decide your evaluation path:
  - **IF no previous feedback was found:** This is a **First-Time Review**. Evaluate the content against the "Guiding Principles Checklist" below. The result of your fact-check from Step 2 is the most important factor for the "Relevance & Accuracy" item on the checklist.
  - **IF previous feedback was found:** This is a **Re-Evaluation**. Your primary task is to confirm if your previous feedback was reasonably addressed.

  4. **Make a Final Decision:**
  - If the slide is clear, accurate, and well-structured according to the checklist, you **must** set `"Slide good enough": true`.
  - If there is a **clear and significant violation** of the checklist (e.g., the content is factually wrong based on your fact-check, or it's a dense paragraph instead of bullet points), set `"Slide good enough": false` and provide specific, actionable feedback.

  5. **Save Your Decision:** You **must** call the **`Save Feedback`** tool with your evaluation results. After successfully saving, move on to the next point.

6. **Final JSON Output:** After saving, your final task is to output the exact same data you just saved, formatted as a single, valid JSON object.

Guiding Principles Checklist (Your Evaluation Standard):
- **Readability & Brevity:** Is the slide easy to scan? Does it use short, concise bullet points instead of long sentences? (Assess the overall feel, do not count words).
- Structure & Splitting:** Does the slide have a clear title (`#`)? Was a complex topic correctly split into multiple, focused slides using `---` to prevent overflow?
- **Relevance & Accuracy:** Does the content match the title and **did it pass your fact-check from Step 2?** This is the most critical check.
- **Feedback Implementation:** (For Re-evaluations only) Was the previous feedback generally addressed?

CRITICAL RULE:
Your job is to stop the loop when a slide is **good enough**. Do not get stuck on minor stylistic details if the slide is fundamentally clear, correct, and readable.

Output Format (Your final response):
```
{
  "Slide good enough": boolean,
  "feedback": "Your specific feedback based on the checklist, or a brief confirmation."
}
```

6. physical view (deployment)

- Workflow engine: n8n workflow engine
- Embedding + LLMs: OpenAI API, OpenRouter Chat Models
- Database: PostgreSQL + PGVector
- Storage: Google Drive
- API: HTTP REST

7. quality requirements

- Maintainability: Modular agent structure
- Scalability: Parallel processing
- Fault tolerance: Evaluation and feedback loops
- Extensibility: New agents can be easily integrated
- Reusability: Embeddings in PGVector
- User-centricity: Manual control + feedback

8. data model (extracted)

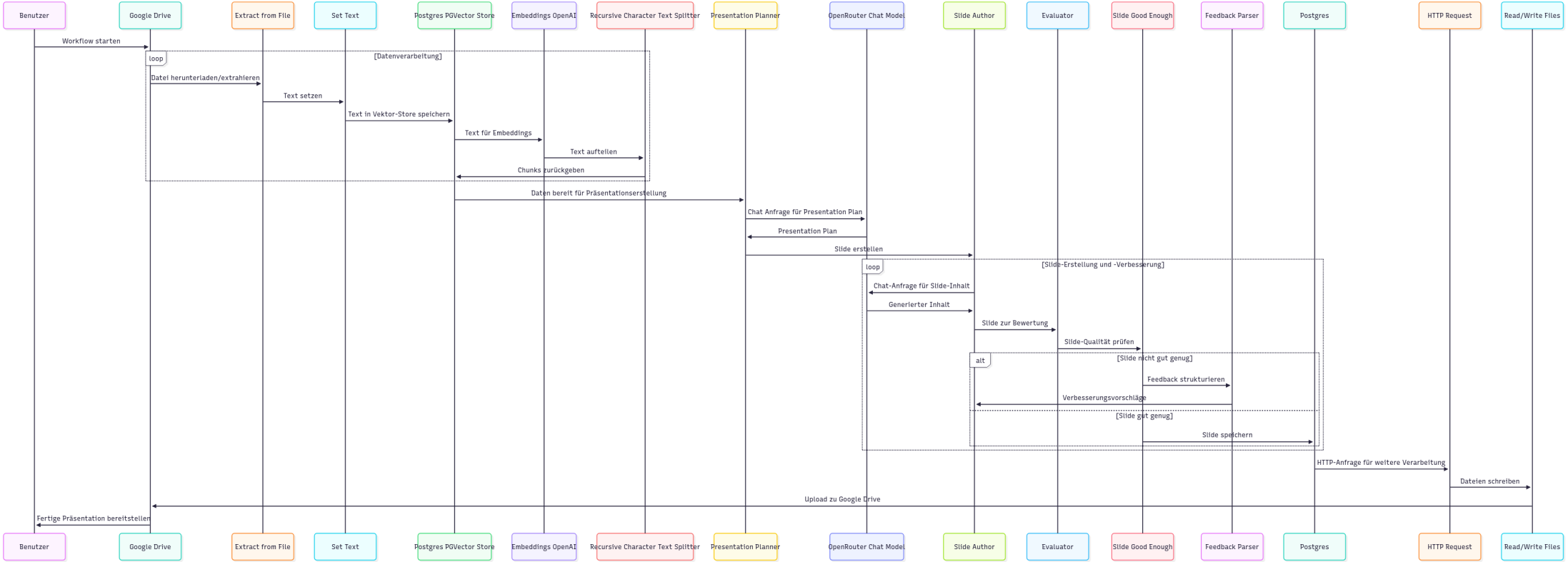
- Presentation: Title, description, metadata
- Slide: title, content, number, status
- Embedding: Vector, source, chunk ID
- Feedback: Score, reason, slide reference
- Evaluation result: JSON-based result

9. potential for further development

- Feedback into the planning process
- Multilingualism
- Integration into e-learning systems
- Real-time co-authoring

10. conclusion

This architecture represents a modern, modular multi-agent system with AI-supported presentation creation. The separation of responsibilities, the use of LLMs and vector databases provide a scalable, adaptive and high-quality solution for the automation of presentation workflows.


Benutzer
Google Drive
Extract from File
Set Text
Postgres PGVector Store
Embeddings OpenAI
Recursive Character Text Splitter
Presentation Planner
OpenRouter Chat Model
Slide Author
Evaluator
Slide Good Enough
Feedback Parser
Postgres
HTTP Request
Read/Write Files
Workflow starten
loop
[Datenverarbeitung]
Datei herunterladen/extrahieren
Text setzen
Text in Vektor-Store speichern
Text für Embeddings
Text aufteilen
Chunks zurückgeben
Daten bereit für Präsentationserstellung
Chat Anfrage für Presentation Plan
Presentation Plan
Slide erstellen
loop
[Slide-Erstellung und -Verbesserung]
Chat-Anfrage für Slide-Inhalt
Generierter Inhalt
Slide zur Bewertung
Slide-Qualität prüfen
alt
[Slide nicht gut genug]
Feedback strukturieren
Verbesserungsvorschläge
[Slide gut genug]
Slide speichern
HTTP-Anfrage für weitere Verarbeitung
Dateien schreiben
Upload zu Google Drive
Fertige Präsentation bereitstellen

⚙️ 1. Technical limits

🔸 Computing resources & performance

- Many slides or papers with many images/diagrams can lead to memory bottlenecks and long runtimes.

- n8n itself is not built for high-performance AI inference - large contexts or parallel GPT calls can become unstable.

🔸 Context window & token limit

- The LLMs (e.g. GPT-4) have limited contexts (128k for GPT-4-Turbo).

- For long papers, the slide author cannot process the entire content, but must be extracted and segmented in advance.

🔸 Error with Base64 / images

- The conversion of PDF images to Base64 is prone to errors (e.g. large files, incorrect coding).

- Dynamically generated images can be difficult to reference or save, especially if no file storage with a public URL is available.

🧠 2. agent logic & autonomy

🔸 No real autonomy

- The agents work sequentially with structured prompts - they do not "understand" the overall system and do not make any real decisions.

- No agent can proactively trigger or coordinate other agents (without you explicitly building this into the workflow).

🔸 No collaborative memory

- Each agent only has access to the context of their current task.

- No "shared memory" or world model: e.g. the Picture Agent does not automatically know what the Planner Agent has designed unless you explicitly specify it.

🧩 3. content limits

🔸 Fact check and source linking

- Agents such as the slide author or evaluator work on the basis of excerpts or summaries - they do not verify statements.

- Without direct access to paper citations, the content can become inaccurate or too generic.

🔸 Image relevance vs. visualization

- Images are usually found via Google or generatively. The content fit is only semantically, not contextually accurate.

- Complex diagrams from the paper are not automatically adopted or reinterpreted unless you build a specialized pipeline for them.

🔒 4. tooling and workflow limits (n8n)

🔸 No built-in long-term memory

- n8n does not store any permanent "memories" - each run starts anew. Agents do not remember what they have done "before".

🔸 Maintenance with many agents

- The more agents, the more complex the maintenance, troubleshooting and prompt control.

- With dynamic requirements (e.g. slides with special layouts or technical jargon), many manual corrections are required.

🧭 Summary: Important limits of your MAS

Area	Limitation
Token Context	Large papers must be split or reduced in size
Autonomy	No real "thinking" or decision-making between agents
Factual Accuracy	No integrated source verification or citation logic from the paper
Image Processing	No OCR or structured text extraction from images
Generated Content	Quality depends heavily on prompt design and intermediate feedback

Area	Limitation
Orchestration (n8n)	No memory, no true scheduling, no self-healing in case of errors
Collaboration	No multimodal feedback or visual collaboration between agents