Degree Project in Computer Science and Engineering

Second cycle, 30 credits

MLPlatformAgentBench

Can coding agents optimize machine learning platform interfaces?

BENEDICT WOLFF

MLPlatformAgentBench

Can coding agents optimize machine learning platform interfaces?

BENEDICT WOLFF

Master's Programme, ICT Innovation, 120 credits
Date: August 17, 2026

Supervisors: Aleksey Veresov, Jim Dowling
Examiner: Amir Hossein Payberah
School of Electrical Engineering and Computer Science
Host company: Hopsworks AB
Swedish title: MLPlatformAgentBench
Swedish subtitle: Kan kodningsagenter optimera gränssnitt för maskininlärningsplattformar?

Abstract

Coding agents are operating machine learning platforms, yet no systematic study has established which platforms they can operate, and at what cost. This thesis maps and measures that landscape.

A market assessment scores 39 machine learning platforms by whether an agent can manage feature, training, and inference pipelines through a Python software development kit, a command line interface, or a Model Context Protocol server. It also examines platform agents and platform meta-harnesses. Five platforms provide complete coverage through at least one of the first two interfaces, and three through both. One of those three is European. None provides full Model Context Protocol coverage.

MLPlatformAgentBench evaluates the interfaces on 22 tasks from feature engineering through operations, restricting each agent to one interface and basic shell tools. Across Hopsworks and Databricks, two American and two European models, two interfaces, and, for Claude, two skill conditions, coding agent and model choice made the largest performance difference. Claude Opus in Claude Code solved 98% of runs, compared with 36% for Mistral Large in Mistral Vibe. The interface changed how work was done, not whether it succeeded. When the command line interface and software development kit were similarly mature, progressive disclosure offered no measurable advantage. Also, skills had no detectable effect on Claude Code's accuracy. Operating platforms differed detectably in accuracy only for Mistral Large in Mistral Vibe, which was more accurate on Hopsworks. The Hopsworks configuration also used fewer turns, less local time, and lower Claude model invocation cost, but the design does not isolate platform from execution period and coding agent version. The mean Claude run cost 1.06 against 2.32 US dollars. Claude Code then produced six optimized command line interface variants and one optimized set of skills for each optimization. None beat its corresponding baseline. This null result supports treating agent-based interface optimizations as hypotheses requiring paired measurement.

Keywords

Machine Learning Platforms, Coding Agents, Large Language Models, Benchmark, Command Line Interface, Software Development Kit, Agent Skills, Sovereign Artificial Intelligence

Sammanfattning

Kodningsagenter använder maskininlärningsplattformar, men ingen systematisk studie har fastställt vilka plattformar de kan använda och till vilken kostnad. Denna avhandling kartlägger och mäter detta landskap.

En marknadsbedömning poängsätter 39 maskininlärningsplattformar utifrån om en agent kan hantera funktions-, tränings- och inferenspipelines via ett Python-programvaruutvecklingskit, ett kommandoradsgränssnitt eller en Model Context Protocol-server. Den undersöker också plattformsagenter och plattformsmeta-harnesses. Fem plattformar ger fullständig täckning genom minst ett av de två första gränssnitten, och tre genom båda. Ett av dessa tre är europeiskt. Ingen ger fullständig Model Context Protocol-täckning.

MLPlatformAgentBench utvärderar gränssnitten på 22 uppgifter från funktionsutveckling till drift, och begränsar varje agent till ett gränssnitt och grundläggande skalverktyg. För Hopsworks och Databricks gjorde två amerikanska och två europeiska modeller, två gränssnitt och, för Claude, två färdighetsvillkor, kodningsagent och modellval den största prestandaskillnaden. Claude Opus i Claude Code löste 98% av körningarna, jämfört med 36% för Mistral Large i Mistral Vibe. Gränssnittet förändrade hur arbetet utfördes, inte huruvida det lyckades. När kommandoradsgränssnittet och programvaruutvecklingspaketet var lika mogna, erbjöd progressiv avslöjande ingen mätbar fördel. Färdigheter hade inte heller någon märkbar effekt på Claude Codes noggrannhet. Operativplattformar skilde sig märkbart åt i noggrannhet endast för Mistral Large i Mistral Vibe, vilket var mer exakt på Hopsworks. Hopsworks-konfigurationen använde också färre varv, mindre lokal tid och lägre kostnad för att anropa Claude-modellen, men designen isolerar inte plattformen från exekveringsperiod och kodningsagentversion. Den genomsnittliga Claude-körningen kostade 1,06 jämfört med 2,32 amerikanska dollar. Claude Code producerade sedan sex optimerade kommandoradsgränssnittsvarianter och en optimerad uppsättning färdigheter för varje optimering. Ingen slog motsvarande baslinje. Detta nollresultat stöder behandling av agentbaserade gränssnittsoptimeringar som hypoteser som kräver parade mätningar.

Nyckelord

Maskininlärningsplattformar, Kodagenter, Stora språkmodeller, Riktmärke, Kommandoradsgränssnitt, Programmeringsgränssnitt, Agentfärdigheter, Suverän artificiell intelligens

Acknowledgments

First and foremost, I would like to thank my late mother, Judith Williams-Wolff, for inspiring me to pursue my master's studies abroad. Born in the United Kingdom to German and British parents, she was the first in her family to attend university and the first to receive a scholarship to continue her bachelor's studies in European Studies in Germany. She accepted it in 1977, only four years after the United Kingdom joined the European Communities, a predecessor of the European Union, and only one generation after my German and British grandfathers fought in opposing forces during the Second World War. Despite her lifelong wish to advance her education, she chose to prioritize a stable income to support her husband, my father, through his PhD and raise my brother and me. Personally, I have always admired her courage, selflessness, and open-mindedness, and I hope that these qualities will continue to guide me in achieving my future goals. Professionally, I feel proud to honor her academic achievements and enduring commitment to the European Union by becoming the first member of my family to pursue a European double master's degree in two member states outside of Germany.

Secondly, I would like to express my gratitude to Aleksey Veresov for supervising this thesis. Although I considered myself reasonably experienced in academic writing before beginning this study, his guidance, meticulous attention to detail, and ability to provide precisely the right amount of feedback whenever needed significantly strengthened my abilities as a researcher and engineer. His mentorship was instrumental in enabling me to publish my first conference paper while completing this thesis.

Thirdly, I would like to thank Jim Dowling for offering me an internship at Hopsworks and for granting me complete academic freedom throughout my time at the company. I still remember approaching him during a break in one of his classes on machine learning systems at KTH to tell him how much I admired his accomplishments as both an academic and an entrepreneur. After introducing ourselves in German, he opened Hopsworks and showed me the problems his team was working on. One walk to the Tekniska högskolan metro station, during which I barely understood the technical details he was referring to, and two half-hour meetings later, I had both an internship and a thesis topic. In addition to machine learning systems, Jim taught me, someone with perfectionist tendencies, the importance of being pragmatic, straightforward, and firmly grounded in research.

I would also like to thank the entire team at Hopsworks for being such intelligent, technically adept, and yet down-to-earth people. In particular, I

am grateful to Manu Joseph, Alexandru Ormenisan, and Javier de la Rúa Martínez for their willingness to help me navigate various aspects of the Hopsworks platform whenever I needed assistance.

Finally, I would like to acknowledge the extensive use of large language models (LLMs), specifically Anthropic's Claude Opus 4.5 through 4.8 and Claude Fable 5, and Open Artificial Intelligence (OpenAI)'s generative pre-trained transformer (GPT)-5.6 Sol, during the preparation of this thesis. These models were used to assist with drafting, editing, refining, and improving the clarity of the written text. They were also used as a tool for discussing ideas, exploring alternative formulations, and supporting the development process where appropriate. Since LLMs are also the subject of this thesis, they served as both a practical instrument and a first-hand engagement with the subject under study. Their purpose was not to replace the intellectual process behind this thesis, but rather to make the most effective use of the latest available technology and the seven months of full-time work invested in it. While Anthropic's and OpenAI's models contributed substantially to the writing process, I take full responsibility for the correctness, validity, and scientific integrity of every result, argument, and conclusion presented in this thesis.

Stockholm, August 2026
Benedict Wolff

Contents

List of Figures

List of Tables

List of Acronyms and Abbreviations

AAAI	Association for the Advancement of Artificial Intelligence
AI	artificial intelligence
AIDE	Artificial Intelligence-Driven Exploration in the Space of Code
ALF	Action Learning From
ALFRED	Action Learning From Realistic Environments and Directives
Amazon S3	Amazon Simple Storage Service
API	application programming interface
AppWorld	Application World
AST	abstract syntax tree
AUC	area under curve
AutoML	automated machine learning
AWS	Amazon Web Services
B	billion
BabyLM	Baby Language Model
BASH	Bourne Again SHell
BDI	belief-desire-intention
bench	benchmark
BERT	Bidirectional Encoder Representations from Transformers
BFCL	Berkeley Function Calling Leaderboard
BM25	Okapi Best Matching 25
CI/CD	continuous integration and continuous delivery
CIFAR	Canadian Institute for Advanced Research
CLAI	Command Line Artificial Intelligence
CLI	command line interface
CNN	convolutional neural network
CoSQA	Code Semantic Question Answering
CSV	comma-separated values
DA	Data Analysis
DAI	distributed artificial intelligence
DataSciBench	Data Science Benchmark
DB	database

DBN	deep belief network
DeepEval	Deep Evaluation
dev	development
DevOps	development and operations
DGM	deep generative model
DL	deep learning
EU	European Union
eval	evaluation
FB	Facebook
FeatEng	Feature Engineering
FS	file system
FTI	feature, training and inference
GAN	generative adversarial network
GB	gigabyte
GenAI	generative artificial intelligence
GEPA	Genetic-Pareto
GPT	generative pre-trained transformer
HopsFS	Hopsworks File System
HTTP	Hypertext Transfer Protocol
HumanEvalComm	Human Evaluation Communication
I-MCTS	Introspective Monte Carlo Tree Search
IBM	International Business Machines
IEEE	Institute of Electrical and Electronics Engineers
JSON	JavaScript Object Notation
KIF	Knowledge Interchange Format
KNIME	Konstanz Information Miner
KQML	Knowledge Query and Manipulation Language
LLaMA	Large Language Model Meta Artificial Intelligence
LLM	large language model
LLM2AutoML	Large Language Model to Automated Machine Learning

MAS	multi-agent system
MCP	Model Context Protocol
MD	Markdown
MDC	Model Diagram Code
MIT	Massachusetts Institute of Technology
ML	machine learning
MLE	machine learning engineering
MLOps	machine learning operations
MLPAB	MLPlatformAgentBench
MMAU	Massive Multitask Agent Understanding
MNIST	Modified National Institute of Standards and Technology
NAS	neural architecture search
net	network
NeurIPS	Conference on Neural Information Processing Systems
NLC2CMD	Natural Language Command to Command
NLP	natural language processing
NumPy	Numerical Python
OLAP	Online Analytical Processing
OpenAI	Open Artificial Intelligence
OpenMLDB	Open Machine Learning Database
OS	operating system
PM2.5	Particulate Matter with an Aerodynamic Diameter of 2.5 µm or less
POSIX	Portable Operating System Interface
RAG	retrieval-augmented generation
RBM	Restricted Boltzmann Machine
ReLU	rectified linear unit
REST	Representational State Transfer
RL	reinforcement learning
RLHF	reinforcement learning from human feedback
RMSE	root mean square error
ROC	receiver operating characteristic
RQ	research question
SaaS	software as a service

SAP	System Applications and Products in Data Processing
SAS	Statistical Analysis System
scikit	Scientific Python Toolkit
SDK	software development kit
SECOM	Semiconductor Manufacturing
SEDAR	Semantic Data Reservoir for Heterogeneous Datasets
SGD	stochastic gradient descent
SOTA	state of the art
SQL	structured query language
SWE	software engineering
TIBCO	The Information Bus Company
TPC-DS	Transaction Processing Performance Council decision support benchmark
TPC-H	Transaction Processing Performance Council ad hoc decision support benchmark
USD	United States Dollar
VAE	variational autoencoder
XGBoost	eXtreme Gradient Boosting
YAML	YAML Ain't Markup Language

Chapter 1

Introduction

This chapter introduces the thesis. It describes how coding agents and machine learning (ML) platforms meet at the platform's interfaces and why the origin of both has become a strategic question. It then states the problem, the research questions, and the goals, presents the research methodology, and delimits the scope. It closes with the structure of the remaining chapters.

1.1 Background

Coding agents based on large language models (LLMs) have moved from generating isolated code snippets to carrying out complete software engineering (SWE) tasks, planning their work, executing tools, reading the results, and iterating until a goal is met. At the same time, machine learning (ML) systems are increasingly built and operated on ML platforms that manage machine learning operations (MLOps) and automated machine learning (AutoML) tasks, organized around abstractions such as feature stores and the feature, training and inference (FTI) pipeline architecture.

An agent can operate an ML platform through the same interfaces a human engineer uses, a Python software development kit (SDK) and a command line interface (CLI), and through interfaces built for agents, Model Context Protocol (MCP) servers, ML platform agents, and skills, plain-text documentation that vendors publish for coding agents. Whether an agent succeeds at platform work, and what that work costs in tokens, turns, and time, plausibly depends on which of these tools it is given. These interfaces differ in how much of themselves they place in the agent's context window before any work begins, a property known as progressive disclosure, which practitioners hold responsible for the efficiency of the CLI

but which has not been measured on platform work.

The question has a geopolitical dimension. Artificial intelligence (AI) has become an instrument of competition between states, which restrict exports of computing hardware, finance national infrastructure, and fund domestic model companies. Sovereign AI strategies consequently treat the origin of models and platforms as a strategic variable. Data held by American providers remains subject to United States law regardless of where it is stored, and this matters for ML platforms in particular because they store an organization's proprietary feature data. Whether European coding agents, models, interfaces, platforms, and skills can match their American counterparts in operating ML platforms is an open empirical question. The next section states the problem that follows.

1.2 Problem

This section states the problem in two steps. The first subsection defines the problem, and the second derives the scientific and engineering issues that are solved to address it.

1.2.1 Problem and Definition

No systematic study exists of which machine learning (ML) platforms a coding agent can operate today, through which interfaces, and across which machine learning operations (MLOps) and automated machine learning (AutoML) tasks. No benchmark measures how the choice of interface and skills affects what an agent accomplishes and what it costs. Analyst reports such as Gartner's Magic Quadrant evaluate whether platforms can manage and build agents, but pay little attention to the interfaces through which a coding agent can operate the platform itself, which MLOps and AutoML tasks those interfaces cover, or how platforms compare on this. Coding agent benchmarks target repository-level software engineering rather than platform operation. This thesis therefore maps the landscape of ML platforms operated by agents and measures, under controlled conditions, the effect of interface, skills, model, and platform on what an agent accomplishes and what it costs. Doing so raises the issues described in the next subsection.

1.2.2 Scientific and Engineering Issues

Measuring what an agent accomplishes on machine learning (ML) platforms raises issues that existing benchmarks for large language models (LLMs) and

ML agents avoid. Tasks must be generated fresh and seeded, so that no solution is already present in model training data, yet remain reproducible. Success must be graded from the state the agent actually produced on the platform to verify that the agent worked on the platform rather than computing results locally. The agent must be verifiably confined to the one interface under test, which requires enforcement rather than instruction. At the same time, different coding agents, models, interfaces, and skills must run under the same conditions and with the same tools, and none of it may restrict the agents' autonomy. Runs must be isolated so that provisioning, teardown, and parallel execution do not contaminate each other. Every reported number must be traceable to the exact run that produced it. Finally, ML platform and model budgets limit the number of runs, so the statistical design must draw its evidence from pairing across tasks rather than from repetition. The next section states the research questions that follow from this problem.

1.3 Research Questions

The purpose of this thesis is to give machine learning (ML) platform vendors, adopting organizations, and the research community an empirical basis for the following three decisions about agent operated ML platforms, each addressed by one of the three research questions (RQs):

- **RQ1:** Which ML platforms exist, and to what extent do their interfaces, namely software development kits (SDKs), command line interfaces (CLIs), and Model Context Protocol (MCP) servers as well as ML platform agents, cover machine learning operations (MLOps) and automated machine learning (AutoML) tasks when assessed against a systematic set of coverage criteria?
- **RQ2:** How do European and American coding agents, based on frontier-tier and lower-tier model products, compare in operating European and American ML platforms across MLOps and AutoML tasks with and without skills, measured by accuracy, model invocation cost, and turns?
- **RQ3:** To what extent can a coding agent optimize an ML platform's CLI and skills, and how does the optimized interface compare against the platform's unoptimized CLI and native SDK baselines on accuracy, cost, and turns?

The economic, social, environmental, and ethical aspects of these research questions are discussed in the reflections in Section 6.4. These are the

cost of agent operated platform work, the sovereignty of the underlying technology, the resource footprint of the experiments, and the confinement of intelligent agents on production systems. The next section defines the goals that follow from these research questions.

1.4 Goals

The goals are five deliverables. First, a market assessment of 39 machine learning (ML) platforms answers RQ1. Second, MLPlatformAgentBench (MLPAB), a benchmark of 22 seeded templates covering feature, training, inference, and operations tasks together with capstone projects. Third, the MLPAB meta-harness for evaluating coding agents operating ML platforms. It wraps each coding agent in a sandboxed, policy-enforced session behind a uniform application programming interface (API), and it executes treatment grids across platforms, models, interfaces, and skill conditions with recorded per-run provenance. Fourth, the executed comparison, 700 graded runs across two platforms, four models, two interfaces, and two skill conditions, answering RQ2. Fifth, the optimization study, in which the agent produces and evaluates six command line interface (CLI) variants and one optimized skill set each, answering RQ3. The next section describes the research methodology by which these deliverables are produced.

1.5 Research Methodology

The thesis follows a quantitative experimental method for software engineering (SWE) research. research question (RQ) 1 is answered by a structured assessment against criteria fixed in advance, and every judgement records a documenting source. The second and third RQs are answered by controlled experiments in which treatments set the independent variables and expand deterministically into run grids, while pinned versions and enforced interface confinement hold the specified factors constant where possible. Analysis uses planned contrast groups evaluated with paired non-parametric tests, Holm adjustment within each group, and effect sizes reported alongside significance. The treatment definitions are retained in the repository, and the test choice for each metric is fixed in the method before the analysis is implemented. The method chapter details the five phases, namely benchmark construction, market assessment, scoping, planning, and operation. The next section delimits the scope of the work.

1.6 Delimitations

Due to financial restrictions, the committed experiments cover two of the five platforms identified as leading, Hopsworks and Databricks, and four models from two vendors, Claude Opus and Claude Sonnet against Mistral Large and Mistral Medium, so conclusions about other platforms and models are out of scope. The Model Context Protocol (MCP) is assessed in the market study but not benchmarked, since no platform covers machine learning operations (MLOps) and automated machine learning (AutoML) tasks completely. Each cell of the design, meaning one combination of platform, model, interface, skill condition, and task, holds a single run, so inference is based on paired contrasts over the 22 tasks rather than on cell level estimates. The tasks are generated rather than collected from production incidents, their scale is bounded by the run budget, and they were not validated by machine learning (ML) experts. The optimization study is limited to the European platform's command line interface (CLI) and a single American model, so the sovereignty comparison does not extend to the optimization results. It also evaluates the optimized artifacts on the task distribution from which they were derived. Security questions, such as whether an unconfined agent escapes its intended interface, are observed only incidentally through the enforcement logs and are not studied. The chapter closes with the structure of the remaining thesis.

1.7 Structure of the Thesis

Chapter 2 provides the background on machine learning (ML) platforms, large language models (LLMs), coding agents, and sovereign artificial intelligence (AI), reviews the related work on meta-harnesses, ML agents and platforms, benchmarks, and agent driven optimization, and closes with a summary that leads to the gaps and derives the research questions (RQs). Chapter 3 presents the method, the construction and verification of the benchmark, the market assessment criteria, and the scoping, planning, and operation of the experiments. Chapter 4 reports the market assessment, the benchmark comparison, and the optimization evaluation, together with their statistical analysis. Chapter 5 discusses the findings, reconstructs from the run artifacts why models underperform, and draws the practical implications. Chapter 6 concludes, states the limitations of the research results, which build upon the delimitations defined above, outlines future work, and reflects on the economic, social, environmental, and ethical aspects of the work.

Chapter 2

Background

This chapter presents the foundational concepts that underpin this study. As the thesis examines machine learning (ML) platforms, it begins with an overview of artificial intelligence (AI) and its subfields, then moves from ML, where the core learning paradigms of modern model training are introduced, to ML engineering, emphasizing practical methods and tools for building models, and further to machine learning operations (MLOps), with its focus on automation and scalable operation of the ML lifecycle. Finally, it introduces ML platforms, the technical infrastructure that supports all of these practices.

The chapter then explores large language models (LLMs) in the context of software engineering (SWE), beginning with deep learning (DL), the subset of ML that enables the neural network architectures behind modern generative models, and generative artificial intelligence (GenAI), the theoretical foundation for models that generate new content rather than merely analyzing existing data. LLMs are examined as the specific class of generative models, followed by code generation as the SWE task to which they are applied, and vibe coding as an emerging development practice that uses these models for SWE tasks. Automated machine learning (AutoML) is then discussed as the application domain in which LLMs are increasingly employed to automate the construction of ML pipelines. Finally, sovereign AI is examined as the geopolitical dimension of these models, establishing why the origin and scale of the LLMs underlying coding agents, and the origin of the ML platforms they operate, constitute strategic variables for this thesis.

As this research focuses on coding agents that use LLMs to optimize ML platform interfaces, it is essential to situate them within the broader field of intelligent agents. For that, intelligent agents and the principles of autonomy, reactivity, and proactiveness that define agent behavior are

examined next. Distributed artificial intelligence (DAI) and multi-agent system (MAS) are discussed to establish coordination and planning principles for environments where multiple agents or agent components collaborate. Software agents are then introduced as the bridge between theory and practical implementation, leading to coding agents, the class of LLM-based AI agents central to this thesis. Context engineering is also covered, as the effectiveness of coding agents depends on how information is structured within their underlying LLM's context window, directly influencing interactions with external tools and platforms.

Building on these foundations, the chapter then examines how coding agents interact with external systems. It reviews the evolution of tool use from early self-supervised approaches to the function calling paradigm, and then examines the three interface types through which agents operate external systems: the software development kit (SDK), the programmatic interface through which a platform is operated from code, the command line interface (CLI), an interface that enables agents to operate directly within the local operating system, and the Model Context Protocol (MCP), the agent-native protocol that attempts to standardize how agents discover and invoke external capabilities. The CLI is examined from its early development in interactive computing systems and early command interpreters, through the Bourne Shell, to modern research on natural language to Bourne Again SHell (BASH) translation. Agent skills are then reviewed as reusable behavioral abstractions that guide agent planning and execution, culminating in agentic coding manifests that provide agents with persistent project context and operational rules. Finally, progressive disclosure is traced from its origin in graphical user interface design to its application to skills, tools, and the CLI, as the principle that determines how much of an interface reaches the model.

The next section reviews related work across four areas, with the aim of identifying gaps in current research that motivate the research questions (RQs). The first traces the harness from its first appearances to its component definition, and fixes the meta-harness definition this thesis adopts. The second examines ML, covering both approaches to specializing software agents for ML workflows and the broader landscape of ML platforms. The third addresses existing benchmarks for coding agents, data and feature engineering, the CLI, and ML agents. The fourth explores existing research on optimizing systems using agents. The chapter closes with a summary that synthesizes the background and related work and derives the RQs guiding this thesis.

2.1 Artificial Intelligence

Artificial intelligence (AI) is a broad term describing "algorithms capable of performing tasks that typically require human intelligence, such as understanding natural language, recognizing patterns, making decisions, and learning from experience" [1]. Early AI systems, including expert systems and knowledge bases, were rule-based and designed to support users and businesses in decision-making processes. Machine learning (ML), a subfield of AI, focuses on algorithms that learn to solve tasks from data without explicit programming. Deep learning (DL), in turn, is a subset of ML that uses neural networks to detect correlations and patterns in large datasets.

While ML and DL primarily focus on analyzing and learning from data, deep generative models (DGMs) aim to capture the underlying distributions of a dataset to generate outputs that closely resemble real-world data [1]. When built on DL architectures, these models are considered generative artificial intelligence (GenAI). Generative modeling allows these models not only to replicate data patterns but also to understand the structure of the information and produce coherent, structured outputs. This capability is particularly effective in domains such as natural language processing (NLP), where models are trained to comprehend and generate natural language. Among the most prominent examples are large language models (LLMs), such as generative pre-trained transformers (GPTs), which apply generative modeling to language and are increasingly used for software code generation. Development practices that rely primarily on these models for code generation are commonly referred to as vibe coding.

2.1.1 Machine Learning

Machine learning (ML) is defined as "a set of methods that can automatically detect patterns in data, and then use the uncovered patterns to predict future data, or to perform other kinds of decision-making under uncertainty" [2]. ML methods are commonly divided into supervised learning, unsupervised learning, semi-supervised learning, and reinforcement learning (RL).

Supervised learning involves learning a function that maps an input to an output based on labeled examples. A training dataset consisting of input–output pairs is used to infer this mapping [3]. In that dataset, each input is represented as a d-dimensional vector of features. Features correspond to a measurable property of the input [2]. For example, in a weather prediction task, a single day could be represented as two features,

the temperature and the day of the year, allowing the model to learn seasonal trends. Supervised learning is usually applied when target outcomes are known. Common tasks include classification, which assigns inputs to discrete categories such as “rainy” or “not rainy”, and regression, which predicts continuous values such as the temperature in degrees.

Unsupervised learning, in contrast, analyzes data without labeled outputs or explicit human guidance [3]. Its objective is to identify patterns, structures, or relationships within the data through techniques such as clustering, dimensionality reduction, and anomaly detection. For example, clustering can group customers with similar purchasing habits, while dimensionality reduction can identify key factors in climate data.

Semi-supervised learning combines labeled and unlabeled data, which is useful when labeled data is limited or costly to obtain [3]. By using the structure of unlabeled data alongside available labels, the aim is to improve predictive performance. Applications include machine translation, fraud detection, and image classification where only a subset of images is labeled.

RL takes a different approach, involving an agent that learns to make decisions through interaction with an environment [4]. The agent receives rewards or penalties based on its actions and seeks to learn a policy that maximizes cumulative reward over time. This approach is well suited for sequential decision-making problems such as robotics, autonomous driving, and game playing, but is less suited for simple prediction tasks.

While these learning paradigms provide the theoretical foundation for ML, applying them effectively in practice requires a set of engineering principles, a structured lifecycle, as well as tools and organizational practices.

2.1.1.1 Machine Learning Engineering

Several practical principles are essential for the effective application of ML [5]. Learning is determined by the combination of representation, evaluation, and optimization, with generalization being the primary measure of success. While large datasets contribute to improved performance, data alone is not sufficient, and feature engineering often has a greater impact than algorithm choice. A central challenge in ML is overfitting, which occurs when a model learns the training data too closely and fails to generalize to new, unseen data [6]. Generalization error decomposes into bias, when a model captures incorrect patterns, and variance, when it captures random noise, and overfitting is primarily a problem of excessive variance, whereas high bias is characteristic of underfitting. Simple algorithms can outperform

more complex ones under computational constraints, and ensembles of models generally achieve better results than individual models. For example, bagging combines multiple decision trees trained on different random subsets of the data to reduce variance, while stacking combines predictions from multiple models to improve overall accuracy.

Understanding these principles is critical not only for building models but also for ensuring that their behavior can be trusted and communicated [7]. Interpretability refers to the extent to which a human can comprehend how and why a model produces its outputs. It can be described along three dimensions. Predictive accuracy measures how well the model approximates the true underlying process. Descriptive accuracy evaluates how faithfully an explanation reflects what the model has actually learned. Relevance considers whether the explanation provides meaningful insights for a specific audience.

The artificial intelligence (AI) lifecycle, synonymously referred to as the ML lifecycle, is usually organized in three phases [8]. The first is the design phase, typically managed by a data scientist, and encompasses identifying and formulating the problem, reviewing data and AI ethics, reviewing technical literature on AI algorithms, applications, and pre-trained models, as well as data preparation, data exploration, and external data acquisition. The second is the development phase, carried out by AI/ML scientists, which involves data pre-processing, building an initial model, augmenting data, developing a benchmark, building multiple models, evaluating primary metrics such as accuracy, precision, recall, and mean squared error, and explaining or interpreting the model. The third is the deployment phase, managed by the AI/ML engineer, and consists of evaluating secondary metrics such as inference latency, throughput, memory footprint, and serving cost, deploying the model and assessing risks, reviewing the deployment, operationalizing the model, and monitoring and evaluating the model's performance.

The most popular programming language in ML is Python [9]. Python is a high-level interpreted programming language, meaning it abstracts away most low-level hardware details, with the reference implementation compiling source code to bytecode that an interpreter then executes, rather than compiling it ahead of time into machine code [10]. Python supports an open-source ecosystem of frameworks for ML engineering, including data preparation, model training, and visualization [9]. Open-source software is software whose source code is publicly available under a license that permits anyone to inspect, modify, and redistribute it, typically subject to conditions specified by the license [11]. Key tools include Numerical Python (NumPy) for array programming and mathematical abstractions [12],

Pandas for data analysis and manipulation, Scientific Python Toolkit (scikit)-learn and TensorFlow as ML libraries [13], NetworkX for graph analytics, and Matplotlib for visualization [14].

Successful ML engineering also requires technical and organizational practices across data, training, coding, deployment, collaboration, and governance [15]. These include validating external data, versioning data, models, and training scripts, automating deployment, monitoring models, and ensuring fairness, accountability, and privacy. Applying them consistently to production ML is the focus of machine learning operations (MLOps).

2.1.1.2 Machine Learning Operations

Development and operations (DevOps) emerged as a paradigm aimed at reducing the gap between development and operations teams by promoting collaboration, communication, and knowledge sharing [16]. Through practices such as continuous integration and continuous delivery (CI/CD), which support continuous testing, quality assurance, monitoring, logging, and feedback loops, the goal of DevOps is to enable fast, frequent, and reliable software releases. Within this paradigm, software engineers are responsible for developing and operating systems, rather than only one of the two. While the operational challenges associated with production ML systems had already been identified in 2015 [17], MLOps subsequently emerged as an ML engineering practice aimed at systematically automating and operationalizing the ML lifecycle, and gained increasing attention in both industry and academic literature around 2020 [18].

MLOps is defined as the process of producing "ML applications based on code, data, and models in small and safe increments that can be reproduced and reliably released at any time, enabling short adaptation cycles" [19]. Systems involved in MLOps can be categorized into three types: ML platforms, ML applications, and ML automation frameworks. ML platforms provide the architecture to operationalize the ML lifecycle through automated ML pipelines. ML applications are custom pipelines designed to serve a domain- or industry-specific purpose, such as drug discovery. ML automation frameworks comprise specialized tools that enable the automation of individual stages within these pipelines [20].

ML automation frameworks support different parts of the ML lifecycle, including data pre-processing, model training, model management, model deployment, and operations and monitoring. Popular frameworks include MLflow, Airflow, Great Expectations, GitHub Actions, Kibana, and

Grafana [21]. MLflow is an open-source platform for experiment tracking, model registry, and lifecycle management [22]. Airflow is an open-source task orchestration platform that schedules and manages complex workflows [23]. Great Expectations is a framework for profiling, validating, and documenting data [24]. GitHub Actions is a component of GitHub, a web-based platform for collaborative software development using Git [25], that automates software development workflows [26]. Git is a version control system that tracks changes to source code, enabling developers to work on a codebase while maintaining a complete history of modifications [27]. Kibana and Grafana are tools for visualizing and analyzing machine logs [28, 29].

As MLOps defines the practices and automation frameworks for operationalizing ML, the effective implementation of these practices depends on the ML platforms that support them.

2.1.1.3 Machine Learning Platforms

ML platforms operationalize the ML lifecycle, so their design reflects how that lifecycle is structured. The ML lifecycle was traditionally organized into two phases, model-centric ML and MLOps. As managing large volumes of data has become increasingly central, this view has been extended into a three-phase lifecycle in which data-centric ML stands as an independent phase preceding the other two [30].

Data-centric ML covers the collection, analysis, and understanding of data, including quality checks, augmentation, labeling, and versioning. Model-centric ML then covers initial model training, hyperparameter optimization, and retraining. Hyperparameter optimization is the systematic tuning of parameters that govern the learning process in order to improve model performance [31]. Examples include the learning rate, which controls the step size the model takes when updating weights during training [32], and regularization strength, which limits model complexity to prevent overfitting [33]. MLOps finally covers model deployment, interpretability, and monitoring.

A core part of monitoring is measuring skewness and detecting drift, which closes the loop back to data-centric ML by triggering the reevaluation of new or changed data to ensure continued model performance. In the context of ML platforms, skew refers to training–serving skew, a mismatch between the data distribution a model was trained on and the distribution it encounters at inference time, which arises from differences in how features are computed across pipelines or from changes in the data itself, and which degrades predictive performance if left undetected [34]. Similarly, drift

refers to gradual changes in data distributions over time [35]. Four types of drift are commonly recognized [36]. Data ingestion drift occurs when the distribution of newly ingested features or labels significantly differs from existing data. Feature drift refers to changes in the distribution of a recent batch of feature data compared to the model's training dataset. Concept drift occurs when the relationship between input features and the target label changes over time, causing the model to lose accuracy. Prediction drift refers to a change in the distribution of a recent range of model predictions compared to predictions on the training dataset.

Given the central role of data throughout the ML lifecycle, ML platforms are fundamentally concerned with data management, aiming to provide highly available infrastructure for managing feature data with unified application programming interface (API) support [36, 37]. An API is a set of rules and protocols that allows different software systems to communicate and interact with each other in a standardized way [38]. One such platform is the Hopsworks Feature Store, whose architecture is shown in Figure 2.1. The platform supports columnar, row-oriented, and similarity search workloads, as well as feature reuse, data transformations, and consistency between feature engineering, model training, and model inference. Columnar storage organizes data by columns, which is efficient for analytical queries that access a few attributes across many rows [36], while row-oriented storage organizes data by rows, which is better suited for transactional operations that access all record attributes [36].

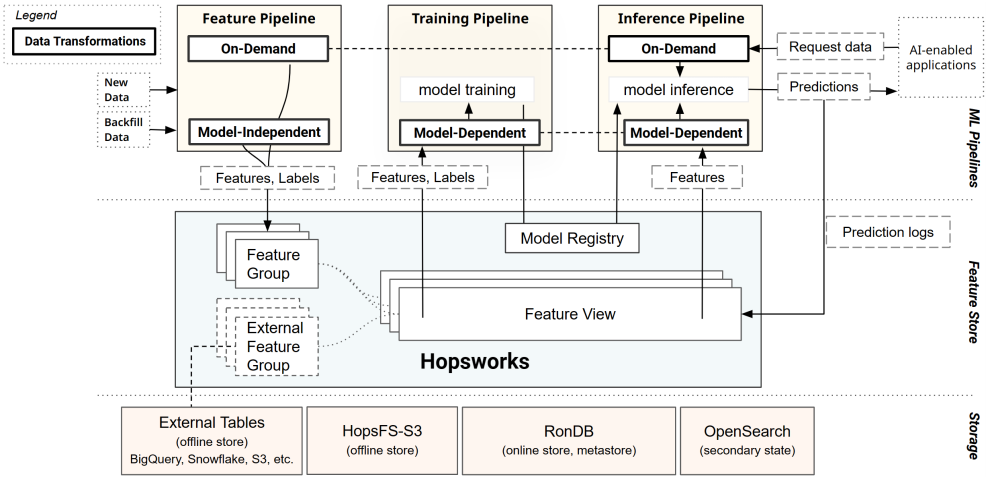

Figure 2.1: The Hopsworks Feature Store for Machine Learning [37]

The system provides centralized access to feature data and organizes ML

systems around three types of pipelines. The feature pipeline transforms input data into features stored in the feature store. The training pipeline reads features and labels to train models and registers them in a model registry. The inference pipeline reads new features alongside a trained model to produce predictions and prediction logs.

Features are organized into feature groups, which are mutable tables of precomputed features, and feature views, which define the schema, filters, and helper columns for a given model. Feature computation is supported in three modes. Batch computations process large volumes of data at scheduled intervals, generating features for subsequent training or inference. Streaming computations continuously process incoming data in real time, updating features as new events arrive. Request-time computations calculate features on-demand at the moment a prediction is requested, ensuring that the model has access to the most up-to-date information.

The platform also distinguishes between model-independent, model-dependent, and request-time features, enabling reuse across multiple models. External data sources can be integrated through services such as BigQuery, which is a serverless data warehouse for large-scale analytics [39], Snowflake, a cloud-based platform for data storage and analytics [40], Amazon Simple Storage Service (Amazon S3), an object storage service for storing and retrieving large datasets [41], the distributed file system (FS) Hopsworks File System (HopsFS) designed for high-performance ML workloads [42], the distributed in-memory database (DB) RonDB optimized for real-time operations [43], and OpenSearch, a search and analytics engine for querying and visualizing structured and unstructured data [44].

Having established how ML models are engineered, operationalized, and supported by platforms, the following section examines deep learning (DL), the subset of ML that underpins modern generative models.

2.1.2 Deep Learning

Deep learning (DL) is a subset of machine learning (ML) that can automatically learn hierarchical representations from raw data, reducing the need for manual feature engineering [45]. The term “deep” refers to the use of multiple successive layers of representation, in contrast to “shallow” models such as decision trees or support vector machines that do not learn through layered transformations. Modern deep networks can consist of hundreds or even thousands of layers, with architectures such as ResNet demonstrating that networks exceeding 1,000 layers can be trained effectively [46]. A typical DL

model consists of an input layer, one or more hidden layers, and an output layer. Each layer is made up of interconnected nodes, where each connection has an associated weight, forming a neural network. Each node computes a weighted sum of its inputs and then produces an activation, which is the output value passed to the next layer. A common non-linear activation function is the rectified linear unit (ReLU), which outputs the input if it is positive and zero otherwise. By applying ReLU, the network can bend and shape its outputs to approximate complex functions in the data, such as detecting edges in an image or predicting patterns in weather measurements.

During model training, the error quantifies the difference between the model's predictions and the desired outputs [47]. The goal is to minimize this error by adjusting the network's weights. Gradients, which indicate how the error changes with respect to each weight, are computed and used to update the weights in a direction that reduces the error. A common optimization method is stochastic gradient descent (SGD), where the model processes small batches of training examples, computes outputs and errors, and updates weights based on the average gradient over the batch. To efficiently compute gradients across multiple layers, backpropagation is used. Backpropagation applies the chain rule of calculus to propagate the error from the output layer backward through the network, allowing gradients for all weights to be calculated efficiently [48].

Convolutional neural networks (CNNs) are a prominent type of DL network [49]. Their structure is inspired by neurons in human and animal brains, and they are widely used in fields such as image recognition and speech processing. Unlike fully connected networks, CNNs use shared weights and local connections to take advantage of the spatial structure in input data. This means only a small number of parameters are needed, which simplifies the training process and speeds up the network. A typical CNN consists of multiple convolutional layers followed by pooling layers and fully connected layers at the end.

In addition to architectures such as CNNs, which are mainly applied to analyzing and classifying data, DL also encompasses models designed to generate new data. This approach is commonly referred to as generative artificial intelligence (GenAI).

2.1.3 Generative Artificial Intelligence

Generative artificial intelligence (GenAI) is defined as "the production of previously unseen synthetic content, in any form and to support any task, through generative modeling" [50]. Early generative models relied on

probabilistic modeling with handcrafted probability distributions to generate synthetic data, but lacked the expressiveness to handle high-dimensional data, such as images, video, or genomic sequences. Foundational implementations included Restricted Boltzmann Machines (RBMs), which are shallow two-layer networks where one layer represents visible inputs and the other captures latent features [51]. Latent features are hidden variables that summarize underlying patterns or structures in the data not directly observable in the input [32]. Deep belief networks (DBNs) stack multiple RBMs to learn progressively more abstract representations [47]. While capable of unsupervised feature learning, both relied on Markov Chain Monte Carlo sampling, a method for approximating probability distributions through sequences of random samples [52], which limited their scalability [53].

A shift toward techniques based on neural networks followed, making GenAI a de facto subtype of deep learning (DL). Variational autoencoders (VAEs) introduced a probabilistic approach using an encoder-decoder architecture, where an encoder compresses input data into a compact latent representation and a decoder reconstructs the output from that representation [54]. This allowed VAEs to model probability distributions in latent space, though they tended to produce blurry samples, limiting their applicability in high-fidelity image synthesis. Generative adversarial networks (GANs) took a different approach, using two neural networks, a generator and a discriminator, competing in an adversarial process [55]. While GANs achieved significant improvements in output quality, they suffered from model collapse, training instability, and sensitivity to hyperparameters, which are configuration settings such as learning rate and batch size that are set before training and influence how the model learns [31]. Subsequent variants addressed these limitations. Deep convolutional GANs incorporated convolutional layers to improve the quality and stability of image generation, while Wasserstein GANs replaced the original loss function with the Earth Mover's distance, a metric that measures the minimum cost of transforming one probability distribution into another [56], which helped stabilize training [53].

In parallel, the field of natural language processing (NLP) underwent significant developments. The introduction of the transformer architecture marked a turning point by replacing recurrent mechanisms, where inputs are processed sequentially and each output depends on the previous step, with self-attention, a mechanism that allows each element in a sequence to directly attend to all other elements simultaneously [57]. This change enabled parallelized training over entire sequences rather than step-by-step processing. A transformer architecture is composed of stacked encoder or

decoder blocks, where each block contains attention layers followed by feed-forward networks. An attention layer produces context-aware representations by allowing each element in a sequence to assign weights to all other elements and combine their information accordingly. A feed-forward network is a neural network layer that transforms each input independently using linear layers followed by a nonlinear activation [58]. Encoder blocks map the input sequence into a set of representations that capture both the meaning of each element and its relationships to other elements in the sequence. These representations are refined across multiple layers of attention and transformation. Decoder blocks generate the output sequentially. At each step, the decoder first applies self-attention over previously generated outputs, and then applies cross-attention over the encoder's full input representation. Not all transformer models use both components. Encoder-only architectures such as Bidirectional Encoder Representations from Transformers (BERT) [59] are designed for understanding tasks, decoder-only architectures such as generative pre-trained transformer (GPT) [60] focus on text generation, and encoder-decoder architectures such as T5 [61] combine both for tasks that require input understanding and output generation. These types of models that rely on large-scale pre-training using large corpora of text are collectively referred to as large language models (LLMs) [62].

2.1.3.1 Large Language Models

LLMs are models capable of executing a diverse range of NLP operations [63]. The most important concepts underlying an LLM include tokenization, attention mechanisms, activation functions, normalization layers, training methods and frameworks, data pre-processing, and parameter tuning. Modern LLMs can have billions (Bs) of parameters, which are the learned weights that determine the model's ability to capture patterns in language [64].

Tokenization refers to the process of parsing text into discrete units called tokens, which can be characters, sub-words, symbols, or words. In the attention mechanism, each token generates three vectors: a query, representing what information the token is looking for, a key, representing what information the token contains, and a value, representing the actual content to be passed forward. Attention scores are computed by comparing each query against all keys, and the output is a weighted sum of the corresponding values. Activation functions such as rectified linear unit (ReLU) introduce non-linearity, while normalization layers maintain stability during training by scaling and shifting activations so that they have a consistent mean and variance, which helps

prevent exploding or vanishing gradients. Training at scale is enabled through parallelism strategies including data parallelism, pipeline parallelism, tensor parallelism, and optimizer parallelism. Data pre-processing involves quality filtering and data de-duplication. Parameter tuning techniques such as prefix tuning, prompt tuning, and adapter tuning allow models to be adapted to specific tasks without retraining all parameters. Beyond these architectural foundations, LLMs can also be improved through human feedback.

Reinforcement learning from human feedback (RLHF) is a technique for aligning model outputs with human preferences and consists of three stages [65]. First, supervised fine-tuning trains the model on prompts paired with desired responses, which can include plain prompts, few-shot examples with query-response pairs, or user-based prompts for specific use cases. Second, a reward model is trained that takes a prompt and response as input and produces a scalar reward score. Third, the model is optimized using a reinforcement learning (RL) algorithm such as proximal policy optimization, where the model generates responses to prompts and receives reward scores that guide its training [66].

In addition to training-time improvements, LLMs support in-context learning, a meta-learning method in which the model uses contextual information provided in a prompt to solve specific tasks without updating its weights [67]. In-context learning takes several forms. Zero-shot learning provides only a task description, while few-shot learning includes examples within the prompt. Chain-of-thought prompting extends this by constructing intermediate reasoning steps that simulate human thinking, improving performance on tasks requiring arithmetic, commonsense, and logical reasoning. Both of these techniques are constrained by the model's context window.

The context window defines the maximum number of tokens an LLM can process in a single forward pass [68]. A forward pass refers to the process in which input data flows through the neural network's layers to produce an output [47], such as predicting the next token. The computational cost of this operation scales quadratically with context length due to the self-attention mechanism, which computes relationships between every pair of tokens in the input sequence. This trade-off must be considered when processing large inputs, as model performance can become increasingly unreliable as the input length grows, a phenomenon referred to as "context rot" in industry [69].

To make effective use of this limited context window, prompt engineering emerged as "the practice of designing and refining input prompts to achieve desired outputs from generative artificial intelligence (AI) models" [70]. Prompt engineering extends in-context learning by maximizing model performance

without adjusting model weights. Over time, entire design philosophies have been developed around prompt construction, such as a problem-oriented, case-driven, and knowledge-complete approach [71]. Problem-oriented means that context is established based on questions posed by technical personnel in real-world scenarios. Case-driven refers to including fault cases for the model to learn from, and adding technical terminology and principles as knowledge points. These problem descriptions and fault cases precede the actual instruction. As a result, accuracy in executing tasks improves by up to 30%.

While LLMs benefit from effective prompt engineering and in-context learning, they also encode and propagate societal biases present in their training data [72]. These biases manifest in two principal forms. Representational harms arise when a model reinforces stereotypes or misrepresents groups, for example by systematically completing "The nurse said that…" with male pronouns despite real-world demographics. Allocational harms occur when model outputs lead to unequal treatment, such as toxicity classifiers producing higher false-positive rates for posts written in specific dialects, resulting in disproportionate content removal. Detecting and measuring such biases requires a layered evaluation approach combining intrinsic methods that probe embeddings and likelihoods, extrinsic methods that assess output-level disparities across tasks, and certification-based techniques that provide statistical guarantees on fairness metrics. Remaining challenges include evaluator bias, limited coverage of non-English languages and intersectional groups, and the risk of over-correction where models achieve fairness by refusing to engage with sensitive topics rather than responding equitably.

Despite these biases and the ongoing challenges in their detection and mitigation, the general-purpose capabilities of LLMs have made them particularly attractive for code-related tasks, leading to their widespread adoption in software engineering (SWE), especially for generating code.

2.1.3.2 Code Generation

Within SWE, LLMs are applied across the entire software lifecycle, spanning requirements engineering, software design, development, quality assurance, maintenance, and management [73]. Among these, code generation and program repair constitute the most prevalent tasks, with decoder-only architectures being the most widely adopted.

The capabilities of LLMs on such tasks are commonly evaluated using SWE-bench, a standardized benchmark that measures a model's ability to resolve real-world SWE tasks [74]. The benchmark consists of 2,294 SWE

problems drawn from real GitHub issues and corresponding pull requests across 12 popular Python repositories. Given a codebase and an issue description, a model is tasked with generating a patch that resolves the issue, with success determined by whether the patched codebase passes the associated test suite. Resolving these issues frequently requires understanding and coordinating changes across multiple functions, classes, and files simultaneously, making SWE-bench a challenging testbed that goes far beyond traditional code generation tasks. Early evaluations showed that the best-performing model at the time, Claude 2, resolved only 1.96% of issues. Since then, progress has been immense. Claude Opus 4.6 achieves 80.8% [75] and GPT-5.2 by Open Artificial Intelligence (OpenAI) achieves 80% on the SWE-bench Verified subset [76].

Despite these growing capabilities, LLMs are prone to several categories of errors in code generation, including misinterpretation of prompts, syntax errors, hallucinated objects or attributes, missing corner cases, incomplete generation, and prompt-biased code where the model excessively relies on provided examples [77]. To mitigate these issues, providing additional context such as example test cases or explicit library versions guides the model toward more accurate output. Users should also test generated code and remain mindful that it may not always be reliable.

A particularly challenging domain is DL, where graph-based structures and implicit semantic constraints differ significantly from those in general-purpose code. For that reason, Deep Evaluation (DeepEval), a benchmark of 100 DL code generation tasks, was constructed to evaluate model performance across three dimensions: syntactic quality, semantic accuracy, and runtime executability [78]. Analysis of failure cases across these dimensions reveals three recurring problems. Models frequently mismanage imports and variables, owing to limited awareness of global context and a tendency to hallucinate DL application programming interface (API) elements. They struggle to organize API calls into coherent model structures when multiple call sequences are required. They are also particularly prone to inter-layer errors such as tensor shape mismatches, which stem from numerical, type, and structural constraints in DL code.

Another concern is robustness, the ability of a model to produce correct output when faced with semantically equivalent but syntactically varied inputs. A study evaluating 25 popular LLMs across four programming languages and 27 semantics-preserving code transformations found that current models still exhibit notable robustness issues, although they outperform earlier pre-ChatGPT code models [79]. The study revealed that instruction-tuned models

are generally more robust than their base counterparts, that higher overall performance correlates with smaller performance drops under transformation, and that models are particularly sensitive to transformations altering program control structure. While techniques such as adversarial training, in-context learning, and code standardization were shown to improve robustness to some extent, each also carried potential negative side-effects, indicating that no single mitigation strategy is yet sufficient on its own.

Nonetheless, LLMs are increasingly integrated into software development workflows through the generation of initial code artifacts from documentation, which can reduce effort in early-stage development, although subsequent manual refinement by developers remains necessary. With the increasing popularity of this practice, a term emerged to describe it.

2.1.3.3 Vibe Coding

The term vibe coding was coined by Andrej Karpathy in early 2025 to describe a style of programming in which developers "fully give in to the vibes" and delegate code writing entirely to LLMs, interacting with the codebase through natural language rather than reading or editing the code directly [80]. It has since been formalized in the academic literature as "development practices that foster intuition, expression, and human-AI collaboration" [81].

Four primary themes shape the practice. Tooling impact concerns tools, workflows, and mindsets related to AI-assisted coding. Developer experience frames software development as intuitive, expressive, and flow-based. Human-AI collaboration highlights interactions such as prompt iteration, co-creative workflows, and task delegation. Cultural and organizational framing examines how vibe coding influences team structures.

Three architectural patterns have been identified in vibe coding [82]. Firstly, conversational code generation involves a natural language prompt followed by LLM processing, code generation, and user feedback. Secondly, multimodal expression processing accepts inputs beyond text, such as sketches, and fuses them into a unified intent model before generating code. Thirdly, context-aware generation analyzes the existing codebase, extracts architectural patterns, interprets user intent, and produces code that aligns with and integrates into the existing system.

Interaction patterns between the user and the LLM also vary [82]. Progressive refinement starts with a high-level description, generates initial code, and iteratively adds constraints and specifications to produce improved versions. Explanation-driven development involves the user explaining

generated code, followed by conceptual analysis and refinement. Hybrid editing combines natural language guidance with direct code review, allowing the user to alternate between describing intent and inspecting implementation.

While vibe coding uses LLMs to automate software development through human-AI collaboration, a parallel line of research builds on their demonstrated capabilities in DL code generation and explores their use within automated machine learning (AutoML), where the goal extends from generating individual code snippets to generating complete machine learning (ML) pipelines.

2.1.4 Automated Machine Learning

Automated machine learning (AutoML) is defined as "the automated construction of a machine learning (ML) pipeline on a limited computational budget" [83]. A complete AutoML pipeline, as illustrated in Figure 2.2, consists of four sequential stages. The first stage, data preparation, encompasses data collection, data cleaning, and data augmentation, ensuring that the raw data is sourced, filtered of noise and inconsistencies, and artificially expanded to improve model robustness. The second stage, feature engineering, transforms the cleaned data into a refined feature set through three complementary processes. Feature selection removes irrelevant or redundant attributes. Feature extraction applies mapping functions to create more compact representations. Feature construction generates new features by combining existing ones. The third stage, model generation, is divided into the search space and the optimization methods.

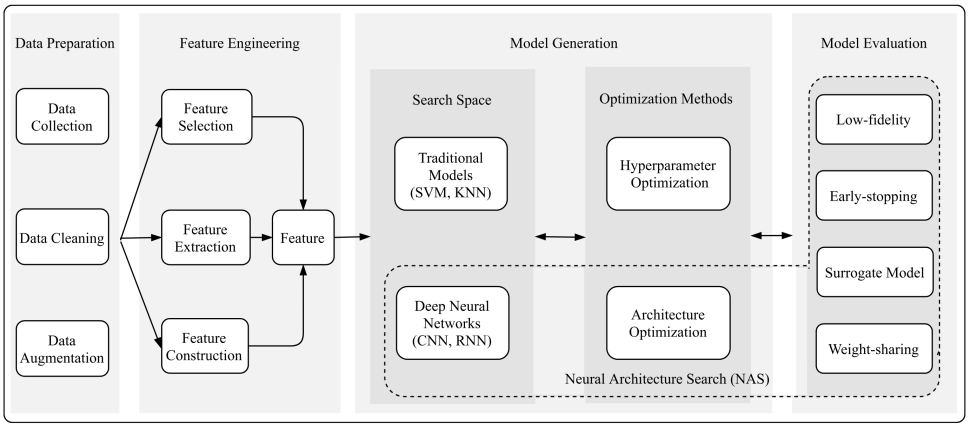

Figure 2.2: Overview of an AutoML pipeline [83]

The search space defines the models that can be explored, ranging from traditional ML models to deep neural networks. The optimization methods are classified into hyperparameter optimization and architecture optimization. Architecture optimization is responsible for tuning model-related parameters such as the number of layers or neurons. When the search space is restricted to deep neural networks and combined with architecture optimization, this sub-field is referred to as neural architecture search (NAS). The fourth stage, model evaluation, determines how well a generated model performs. Several acceleration strategies are used to reduce the computational cost of training every candidate to full convergence. These include low-fidelity evaluation, early stopping, which halts training when performance on a validation set ceases to improve [84], surrogate models, which approximate the objective function to reduce the number of expensive evaluations [85], and weight sharing, which allows candidate architectures to reuse learned parameters rather than training from scratch [86]. Throughout the pipeline, model generation and model evaluation operate in an iterative feedback loop. Candidate models are generated, evaluated, and refined until the best-performing pipeline is identified.

The integration of large language models (LLMs) into AutoML has led to a new generation of ML systems that use natural language understanding to reduce manual effort across the ML pipeline. An early example is Aliro, a platform that combines LLMs with a built-in recommender system for the selection of ML algorithms and hyperparameters [87]. Aliro supports chat-based interaction, code editing, package installation, data visualization, and data processing. Users provide an Open Artificial Intelligence (OpenAI) application programming interface (API) key to enable LLM-driven code generation, and the system offers the ability to save artifacts produced by scripts, such as text and image files. Its key contribution is the LLM-based recommender system, which guides users through model building without requiring deep expertise, as demonstrated in a use case involving Alzheimer's disease research using a random forest classifier, an ML algorithm used for classification tasks [88].

Building on this direction, Large Language Model to Automated Machine Learning (LLM2AutoML) proposed a zero-code AutoML framework that integrates LLMs across intention understanding, model recommendation, and hyperparameter optimization [89]. The framework employs prompt engineering for intention extraction, in which the LLM assumes the role of an experienced ML expert, with prompts refined through iterative feedback cycles. Key capabilities include feature extraction with recursive feature

elimination to identify salient features, meaning those that contribute most strongly to the predictive performance of a model [90, 91], as well as automatic ablation studies and automated hyperparameter optimization. The framework also introduces adaptive focal loss as an alternative to traditional cross-entropy loss, a loss function that measures the divergence between predicted probability distributions and actual class labels and is widely used as the default objective in classification tasks [32]. Report generation is also automated. The framework was evaluated on the Semiconductor Manufacturing (SECOM) dataset to emulate semiconductor smart manufacturing, comparing OpenAI o1-mini and Claude-3.5-Sonnet. Results showed that LLM-recommended hyperparameter spaces improved model performance, particularly in convergence speed and area under curve (AUC) values, a metric that quantifies how well a classifier distinguishes between classes by measuring the area under the receiver operating characteristic curve [49]. This demonstrated strong generalization capabilities and expert-level tuning performance when integrated with search algorithms.

A systematic survey then provided the first software engineering (SWE)-oriented, stage-wise mapping of LLM contributions across the full AutoML workflow, analyzing 34 studies published after 2022 [92]. The survey organizes contributions into three stages. In the first stage, data and feature engineering, LLMs support data acquisition through dataset recommendation and summarization. For data cleaning, they enable adaptive imputation, which dynamically selects strategies for filling in missing values based on contextual cues in the data, and context-enhanced transformation, which uses surrounding semantic information to decide how data values are reformatted or standardized. Data augmentation is supported through both modification of existing samples and synthetic data generation. For feature engineering, LLMs are employed across feature selection, extraction, and synthesis, with frameworks integrating these into pipelines that enable dynamic operator composition, the automated selection and chaining of feature transformation operations at runtime. In the second stage, model selection and hyperparameter optimization, retrieval-based methods match task descriptions against curated model repositories to recommend candidate algorithms, while generation-based methods synthesize selection logic or entire architectures directly from natural language. For hyperparameter optimization, execution-based methods iteratively refine configurations by training models and observing real performance feedback, while prediction-based methods bypass full training by using surrogate models to simulate outcomes. In the third stage, workflow evaluation, agent-based methods

deploy LLM-powered agents that predict performance from task descriptions or partial executions without completing full training cycles, while proxy-based methods employ lightweight surrogate models or zero-cost heuristics, approximation strategies that estimate metrics without exhaustive computation [93], to approximate key metrics in a single forward pass.

Across all stages, the survey identifies persistent challenges around transparency, correctness, and robustness, and highlights that production-grade adoption demands maintainability, interpretability, and traceability of generated logic. Among its future research directions, the survey calls for embedding ML workflow generation into full-stack data applications. It argues that workflows should integrate upstream with data lakes, catalogs, and schema managers to guarantee provenance and compliance, and downstream with dashboards, visualization layers, and application-specific front-ends to support enterprise decision-making. A data lake is a centralized repository that stores large volumes of raw, structured, semi-structured, and unstructured data in its native format, allowing for flexible analytics and processing [94]. This would allow LLM-generated AutoML workflows to evolve from research prototypes into production-grade systems.

In addition to technical challenges, the deployment of AutoML in production environments introduces important considerations related to data sovereignty, governance, and regulatory compliance. These concerns underpin the concept of Sovereign artificial intelligence (AI).

2.1.5 Sovereign Artificial Intelligence

Sovereign artificial intelligence (AI) refers to a nation's capacity to develop, operate, and control AI systems using domestic infrastructure, data resources, workforce expertise, and commercial ecosystems, independent of foreign platforms or corporations [95]. The concept gained prominence around 2020 within European policy discussions on digital sovereignty and data protection, and its visibility was subsequently elevated by NVIDIA, whose chief executive began promoting the term in late 2023. This contemporary usage should be distinguished from an earlier one in the AI safety literature, where sovereign AI denoted a hypothetical super-intelligent system acting independently on a global scale [96]. Nations pursue AI sovereignty for several reasons [95]. National security is one, since reliance on foreign systems can introduce vulnerabilities into critical infrastructure. Data privacy and control is another, since domestically held data is shielded from foreign jurisdictions. Economic competitiveness follows from the cultivation of a local AI ecosystem, and

regulatory autonomy concerns standards of transparency and accountability. Cultural and linguistic representation matters because models trained on local data better support local languages and knowledge, and strategic autonomy in times of geopolitical conflict completes the list.

The competition for sovereign AI has been theorized as a new phase of state formation [97]. Building on the classical argument that enduring external pressures generate domestic imperatives for capacity building, the Generative AI-Making and State-Making framework argues that elites' perception of trans-boundary AI competition drives states to strengthen four capacities. Extractive capacity is the ability to mobilize capital, data, and material resources for an autonomous AI production system. Coercive capacity is the authority to deploy non-market mechanisms such as export controls and directive industrial policy. Delivery capacity is the ability to coordinate stakeholders and provide public goods such as research networks and AI education. Informational capacity is the ability to collect and analyze data on technologies and global competition to support policy decisions. The framework predicts that the sharper the perceived rivalry, the greater the strategic investment, and that while states converge in strengthening these capacities, their objectives diverge according to their position in the international system, their geopolitical pressure, and their domestic endowments.

A comparative analysis of the United States, France, Brazil, and Singapore substantiates this mechanism [97]. The United States, perceiving a direct challenge from China, pursues explicit technological hegemony through systematic export controls on advanced chips and through state-led financing such as the $500 billion Stargate initiative. France, by contrast, frames its strategy as a third way between the United States and China. Following an early warning that the country risked becoming a data colony of the two superpowers, it has shifted from market-oriented policy toward state-directed cultivation of national champions such as Mistral AI, backed by a 109 billion Euro investment plan explicitly framed as France's answer to Stargate [98]. Brazil and Singapore illustrate that the race extends beyond the great powers. Both link AI to sovereignty and pursue regional leadership by developing large language models (LLMs) tailored to Portuguese and to Southeast Asia's languages and cultures, respectively. The resulting landscape is characterized by United States–China bipolarity alongside rising regional technological powers, with LLMs increasingly aligning geopolitical competition with linguistic and ecosystem divides rather than purely territorial ones.

Achieving sovereignty in practice requires control over an interdependent stack of resources, comprising energy and physical infrastructure, data and

cloud platforms, models and algorithms, financing, supply chain governance, and human capital, where the capability of the whole is only as strong as its weakest link [99]. At the model layer, sovereignty includes owning the fundamental model artifacts, including the trained weights, the architecture, the training data, and the surrounding software environment, which protects against capability suspension, the risk that an external provider abruptly restricts access due to political pressure or commercial shifts [99]. Measured against these requirements, full-stack sovereignty is attainable only for the United States and China [95]. The chip market is dominated by NVIDIA, with roughly 80% of the global market for AI accelerators, and the data center market by American hyperscalers, led by Amazon Web Services (AWS), Microsoft Azure, and Google Cloud Platform. Moreover, data held by American providers remains subject to extraterritorial United States law regardless of where it is physically stored. Every other nation must therefore settle for partial sovereignty, whose most attainable dimension is the domestic development of culturally and linguistically appropriate models, a sovereignty that is nonetheless exercised "at the whim of external forces" [95].

Within Europe, this partial sovereignty has produced substantial activity at the model layer. The European Union (EU) appointed its first commissioner for technological sovereignty in 2024 and launched the InvestAI initiative, mobilizing 200 billion Euro for AI development, including five gigafactories dedicated to training next-generation models [95]. France hosts Mistral AI, the most competitive AI actor outside the United States and China, valued at $20 billion [98], as well as LightOn, Europe's first publicly listed generative artificial intelligence (GenAI) startup [97]. At the EU level, the OpenEuroLLM collaboration of 20 organizations aims to develop open-source models covering all European languages by 2028. European frontier models have thus emerged as deliberate alternatives to their American counterparts, positioned to serve organizations for which data residency, extraterritorial legal exposure, and the risk of capability suspension are strategic rather than technical concerns.

For organizations that rely on LLMs to automate software engineering (SWE) and machine learning (ML) workflows, the origin of the underlying model is therefore a strategic variable, and so is the origin of the ML platform on which these workflows operate, since the platform stores the proprietary feature data on which concerns over data residency and extraterritorial legal exposure center. Model tier constitutes a further practical variable, as sovereign AI strategies emphasize compact, resource-efficient models deployable under constrained budgets [99]. Whether European and lower-tier

model products can match American frontier-tier products on such workflows, and whether this holds across platforms of either origin, is an open empirical question, motivating the comparative dimension of RQ2.

The models at the center of this competition do not automate such workflows on their own. They are part of LLM-based AI agents equipped with specialized skills and tools, so the strategic questions of origin and tier extend from the model to the agent built around it. The following section examines these agents.

2.2 Agents

The concept of agents has a long history in artificial intelligence (AI), originating in research on rational decision-making and autonomous behavior. Early work formalized how individual agents perceive their environment, form beliefs, and act on intentions. As these ideas matured, attention shifted toward systems of multiple agents operating within shared environments, from which distributed artificial intelligence (DAI) and multi-agent system (MAS) emerged, concerned with coordination, planning, and strategic interaction.

The translation of these theoretical foundations into software led to software agents, autonomous programs designed to operate continuously within computational environments. While early software agents were generally fragile and special-purpose, the convergence of agent architectures with large language models (LLMs) has produced a new generation of coding agents capable of interpreting developer goals, generating code, and interacting with external tools. As these agents grow more capable, their effectiveness increasingly depends on the context they receive and how efficiently it is managed, motivating the examination of context engineering, and, once the tools and skills available to an agent have been introduced, of progressive disclosure, the principle that determines how much of each of them occupies that context.

2.2.1 Intelligent Agents

Agents originated in research on rational decision-making and autonomous behavior. A rational agent's behavior is governed by three mental attitudes, known as the belief-desire-intention (BDI) model [100]. Beliefs represent the set of worlds the agent considers possible at a given point in time, referred to as belief-accessible worlds. Desires represent the set of worlds the agent would like to bring about, known as goal-accessible worlds. Intentions are

the subset of desires that the agent has committed to realizing, which define intention-accessible worlds. Decision-making emerges from comparing belief-accessible worlds and selecting goals that are consistent with these beliefs, such that for every belief-accessible world there exists a corresponding goal-accessible world, although the converse does not necessarily hold. Building on this formalization, the concept can be further specified in terms of the observable properties that characterize intelligent agents in practice.

In information technology, an intelligent agent is most often a software system characterized by autonomy, social ability, reactivity, and proactiveness [101]. Autonomy is the ability to operate without direct human interaction. Social ability is the ability to interact with other agents. Reactivity is the ability to perceive the environment and respond to changes that occur in it. Proactiveness means that agents do not simply respond to their environment but exhibit goal-directed behavior. Agents can also be described as intentional systems, where a first-order intentional system has beliefs and desires, but no beliefs and desires about beliefs and desires. A second-order intentional system has beliefs and desires about beliefs and desires.

Early applications of intelligent agents spanned a range of domains [102]. In manufacturing and logistics, agents were applied to traffic management, flexible manufacturing, and plant control using modular architectures based on behavior, resources, and intentions. In telecommunications, agent-based systems addressed network control, service management, transmission switching, and network design. On the internet, agents were deployed for filtering, evaluating, and integrating information, often operating in teams depending on user tasks and situational knowledge. Further applications emerged in knowledge discovery and information analysis across large data spaces, as well as in financial services, including mortgage sales, corporate financial management, and portfolio management.

While the properties of autonomy, reactivity, and proactiveness characterize individual agent behavior, many of the applications described above involve multiple agents interacting within a shared environment, so-called multi-agent system (MAS) studied in the field of distributed artificial intelligence (DAI).

2.2.2 Distributed Artificial Intelligence

Artificial intelligence (AI) has traditionally focused on the capabilities of individual systems, but many real-world problems are inherently distributed in nature, involving multiple entities that must operate across different locations,

manage separate sources of information, and pursue potentially conflicting objectives [103]. Distributed artificial intelligence (DAI) emerged as a field to address these challenges by combining principles from AI with distributed computing. DAI encompasses two main areas: distributed problem solving, which involves the decomposition and distribution of a problem-solving process among subordinate nodes, and multi-agent system (MAS), which focuses on the joint behavior of agents with some degree of autonomy.

A MAS consists of multiple individual agents that interact with each other through communication and can act within a shared environment [104]. Different agents may have different spheres of influence and control over different parts of the environment. Various relationships can exist between agents, including coincidence, dependency, power relationships, and hierarchies. Agents are assumed to be self-interested, possessing their own preferences and desires. More specifically, dependency relationships between agents can take several forms: independent, where no dependence exists between agents, unilateral, where one agent depends on another, mutual, where both agents depend on each other with respect to the same goal, and reciprocal, where the first agent depends on the second for one goal and the second depends on the first for another.

These relationships shape how agents reason about outcomes. Preferences over outcomes are represented through utility functions, and the actual outcome depends on the combination of actions performed by all agents [104]. Game-theoretic concepts such as dominant strategies and equilibria provide the foundation for analyzing such strategic interaction, although the agents examined in this thesis operate individually rather than strategically.

Coordination in MAS requires a structure within which agents can interact in predictable ways, flexibility so that agents can operate in dynamic environments, and appropriate knowledge and reasoning capabilities to intelligently use both the structure and the flexibility [105]. Commitments produce predictability, which agents need to reason about, while conventions allow agents to respond flexibly to changing circumstances. Coordination can thus be understood as a distributed goal search problem.

Closely related to coordination is the problem of planning, which determines how agents structure their actions toward shared objectives [106]. Planning in MAS involves both individual plans and shared plans. A shared plan for group action incorporates individual and mutual beliefs, and can take several forms: full individual plans, partial individual plans, full shared plans, partial shared plans, and shared plans of indefinite completeness. A collaborative plan for action includes a mutual belief of a partial recipe for

the action, individual intentions that the action be done, individual intentions that collaboration succeed in performing the constituent sub-actions, and individual or collaborative plans for the sub-actions.

While coordination, planning, and strategic interaction provided a theoretical foundation for MAS, translating these ideas into working software required addressing practical concerns such as continuous operation, communication standards, and integration with computing infrastructure. This led to the emergence of software agents as a distinct area of research.

2.2.3 Software Agents

With the emergence of intelligent agents, research also focused on how to integrate agents into software engineering (SWE) processes. A software agent is defined as "a software entity which functions continuously and autonomously in a particular environment, often inhabited by other agents and processes" [107]. Such an agent can operate continuously over a long period of time, learn from experience, and coexist with other agents and processes within a computer system.

Distinguishing software agents from other agent types, one taxonomy categorizes intelligent agents into biological, robotic, and computational agents, with computational agents further divided into software agents and artificial life agents. Software agents are further classified into task-specific agents, entertainment agents, and viruses [108]. An alternative, more extensive classification identifies seven types of software agents: collaborative agents, interface agents, mobile agents, information agents, reactive agents, hybrid agents, and heterogeneous agent systems [109]. Collaborative agents emphasize autonomy and cooperation with other agents to solve problems too large for a single agent. Interface agents emphasize autonomy and learning to perform tasks on behalf of their owners, comparable to a personal assistant, by observing and imitating the user, receiving positive and negative feedback, receiving explicit instructions, and consulting other agents for advice. Reactive agents do not possess internal models of their environment, instead acting in a stimulus-response manner without a priori specification of behavior. Hybrid agents combine philosophies from two or more agent types, while heterogeneous agent systems integrate at least two agents belonging to two or more distinct agent classes.

Building on these classifications, agent-based SWE emerged as an approach to modelling, designing, and implementing computer systems using software agents [110]. The agent-oriented approach advocates decomposing

problems in a way that allows agents to engage in flexible, high-level interactions. Each agent maintains its own thread of control and decides autonomously which actions to perform. This supports the engineering of complex distributed systems in several ways. Agent behavior can be based on the agent's actual local state rather than an external entity's perception of it. It is not necessary to know all potential interactions in advance. Agents are also designed to handle unanticipated requests and to spontaneously generate requests for assistance. If unexpected events occur, agents have the autonomy and proactiveness to try alternative actions. Because schedules are built up dynamically through flexible interactions, they can be readily adapted in the event of delays or unforeseen contingencies. Additionally, explicitly defined relationships between system components identify which agents need to coordinate their actions.

Alongside these developments, efforts were made to establish universal communication languages for agents. Two approaches emerged: a procedural approach, based on the exchange of sequential directives, and a declarative approach, based on the exchange of assertive statements [111]. The procedural approach was limited by its dependence on knowledge of the recipient and its unidirectional nature. The declarative approach gave rise to agent communication languages such as the Knowledge Interchange Format (KIF) and the Knowledge Query and Manipulation Language (KQML).

However, both agent communication languages failed to achieve widespread adoption and have largely fallen out of use. Early software agents were similarly constrained, generally fragile and special-purpose in nature [107]. Software agents have nonetheless seen a resurgence in recent years. Today, coding agents combine the principles of intelligent agents with the ability to generate, execute, and maintain code using large language models (LLMs).

2.2.3.1 Coding Agents

The convergence of LLMs and software agents has led to a new generation of LLM-based artificial intelligence (AI) agents [112], hereafter simply referred to as agents. These agents typically consist of four components as depicted in Figure 2.3: planning, memory, perception, and action.

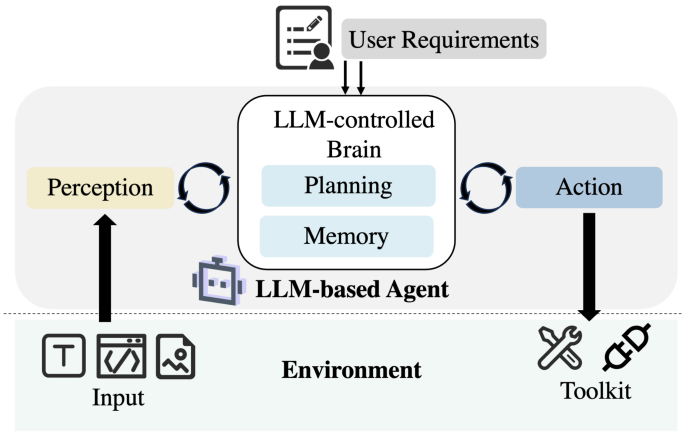

Figure 2.3: Basic framework of an AI agent based on an LLM [113]

Planning and memory serve as the LLM-controlled brain, while the agent interacts with its environment through perception and action to achieve specific goals. The planning component decomposes complex tasks into subtasks and schedules them accordingly, generating or adjusting plans as needed. Memory records historical thoughts, actions, and environmental observations. Perception handles multi-modal inputs such as textual, visual, or auditory signals, and the action component executes concrete operations to interact with and influence the environment, including the control and use of external tools that extend beyond the inherent capabilities of LLMs. User requirements are communicated to the agent's LLM-controlled brain in natural language, that is, through vibe coding.

Based on this foundational understanding, coding agents are loosely defined as agents that, rather than merely executing predefined technical instructions, participate in interpreting a software developer's explicit or implicit goals, anticipate potential needs, and provide personalized outputs based on usage patterns [114]. Most significantly, these agents are capable of offering solutions that may exceed the developer's own knowledge. Popular examples of coding agents include Claude Code [115], which is closed-source, GitHub Copilot [116], which is proprietary with some open-source components, and OpenCode, an open-source alternative [117].

Common use cases for coding agents include automated program repair, where traditional approaches rely on fault localization, semantic analysis, or validation [118]. Coding agents can also automate version management tasks such as upgrading dependencies, fixing compatibility issues, addressing deprecated application programming interfaces (APIs), and handling syntax changes. Additionally, coding agents support quality assurance and code

review prior to human evaluation by suggesting optimizations, detecting inconsistencies, and validating compliance.

Multiple coding agents can also collaborate within a multi-agent system (MAS), where each agent assumes a specialized role in the development process. One example in this domain is automated dependency upgrading, implemented through a system comprising a summary agent, a control agent, and a code agent [119]. The summary agent generates code summaries at varying levels of granularity. The control agent identifies relevant code units by first examining file-level summaries and then drilling down to finer-grained representations, while also consulting migration guides when available. The code agent receives the task along with the selected code units, implements the required changes, and updates dependencies and configurations. After each edit cycle, the project is compiled and tests are executed. If errors occur, logs are returned to the control agent for further resolution, forming an automated program repair loop. This process continues until the build succeeds or a maximum retry threshold is reached. At that point, the framework supports handover to human developers with a summary of the actions taken.

Empirical evidence suggests that coding agents are already contributing meaningfully to real-world software projects [120]. Like human-written pull requests, agentic ones primarily address bug fixes and feature development, though they more frequently involve refactoring, documentation, and test improvements, tend to serve multiple purposes within a single submission, and include more code additions. Agentic pull requests achieve an 83.8% acceptance rate with review times comparable to human contributions, though still below the 91.0% rate for human-written ones, with most rejections stemming from project context, such as overlapping changes or excessive size, rather than flaws in the generated code. Nevertheless, 45.1% of merged agentic pull requests required reviewer revisions, highlighting that while AI-generated code provides a strong starting point, human oversight remains essential to ensure correctness and adherence to project standards.

As coding agents continue to advance, their effectiveness increasingly depends not only on their internal reasoning capabilities but also on the context they receive and how efficiently it is managed within the constraints of the context window. This growing need to optimize what information is presented to agents, and how, motivates a closer examination of context engineering.

2.2.3.2 Context Engineering

Especially in domains that require large volumes of textual input, such as coding agents, techniques have been discovered that extend in-context learning and prompt engineering by optimizing external context before adding it to the model [121]. For example, in code completion, providing top-k results to a coding agent after ranking source code using a keyword-based search algorithm such as Okapi Best Matching 25 (BM25) can yield a 3% improvement using Kotlin and a 9% improvement using Python. File ordering also affects performance, as placing the most relevant files closest to the cursor position and the least relevant files at the beginning of the context ensures that when the context window is exceeded, the agent truncates from the left, discarding the least useful files first, resulting in a 3% improvement for Python. Chunking strategies similarly influence results. Standard chunking ignores import statements and segments code into objects, classes, standalone functions, expressions, statements, and docstrings, and achieves a 6% improvement over the best file-retrieval strategy and a 16% improvement over the no-context baseline for Python. Method-level chunking instead splits code into individual methods, but performs worse than standard chunking for Python and similarly for Kotlin. Lastly, local scope context trimming of prefix and suffix achieves a 3% improvement in Kotlin.

With such techniques in mind and as LLMs became the foundations of modern software agents, prompt engineering evolved into context engineering, defined as "the architectural design of AI systems in which model behavior is shaped by dynamically constructed context beyond the immediate user prompt, including persistent memory, external knowledge retrieval, tool invocation, and policy enforcement mechanisms" [70]. What distinguishes context engineering from prompt engineering is its focus on integrating external tools in addition to user requirements and the model's own perception, as well as using memory mechanisms to achieve continuity across interactions.

One of the most prominent mechanisms to achieve such continuity is retrieval-augmented generation (RAG). In a typical RAG pipeline, documents are first segmented into passages, encoded as dense vector representations [122], and then stored in a vector database [123]. When a query is received, a retriever encodes the query into the same embedding space and retrieves the top-k most similar passages using approximate nearest neighbor search. These passages are then concatenated with the original query into a prompt, and an LLM generates a response conditioned on both the query and the retrieved context. By grounding responses in retrieved documents,

RAG reduces hallucinations [124], model outputs "that appear nonsensical or unfaithful to the provided source content" [125], and enables models to access knowledge that was not present during training.

Alternatives to RAG promise increased accuracy in retrieving external context and reduced token cost. One such approach is Mem0, a memory framework that operates through two phases, namely extraction and update [126]. During extraction, salient facts are identified from new message pairs using a conversation summary and recent message history as context. During the update phase, each extracted fact is compared against existing memories using semantic similarity, and an LLM determines whether to add, update, delete, or ignore it. In its base configuration, Mem0 stores memories as natural language text indexed by vector embeddings. An extended variant, Mem0^{g}, additionally represents memories as a directed labeled graph in which entities serve as nodes and relationships as edges, enabling relational reasoning across interconnected facts.

Whereas retrieval and memory govern what enters the context window, tools determine how the model acquires information and acts upon it, extending context engineering from a curatorial task into an interactive one.

2.2.4 Tools

The ability of large language models (LLMs) to use external tools through application programming interfaces (APIs) has emerged as a key research direction for overcoming inherent model limitations such as arithmetic errors and outdated factual knowledge. Toolformer was the first approach to demonstrate that tool use can be learned in a self-supervised way, without requiring large amounts of human annotations [127]. The approach represents API calls as linearized text sequences embedded within the token stream, delimited by special tokens that mark the start of a call, the arguments, the returned result, and the end of the call. At inference time, the model performs auto-regressive decoding, the standard process by which a language model generates text one token at a time, with each new token conditioned on all previously generated tokens [128]. This proceeds normally until the model generates a special token indicating that it expects a response from an API, at which point decoding is paused, the host system parses the API name and arguments from the generated tokens, executes the call against the appropriate external tool, injects the result into the sequence, and resumes decoding. By incorporating tools such as a calculator, a question-answering system, a search engine, a translation system, and a calendar, Toolformer achieved substantially

improved zero-shot performance across a variety of downstream tasks with a 6.7 billion parameter model, often reaching performance competitive with much larger models without sacrificing its core language modeling abilities.

Building on this foundation, Gorilla addressed the challenge that LLMs remain largely unaware of which APIs are available and how to use them in a frequently updated tool set [129]. Gorilla is a fine-tuned Large Language Model Meta Artificial Intelligence (LLaMA) model that surpasses the performance of generative pre-trained transformer (GPT)-4 on writing API calls. Trained using retriever-aware training, Gorilla demonstrates the ability to adapt to test-time document changes, allowing flexible user updates or version changes, while substantially mitigating hallucination. To evaluate the model, the authors introduced APIBench, a dataset consisting of HuggingFace, TorchHub, and TensorHub APIs.

To bridge the gap between the limited tool-use capabilities of open-source LLMs and the strong tool-use performance of closed-source models such as ChatGPT, ToolLLM introduced a general tool-use framework encompassing data construction, model training, and evaluation [130]. The framework centers on ToolBench, an instruction-tuning dataset constructed automatically using ChatGPT from 16,464 real-world Representational State Transfer (REST)ful APIs spanning 49 categories collected from RapidAPI Hub. RESTful APIs follow an architectural style, in which clients interact with server resources through Hypertext Transfer Protocol (HTTP) methods such as GET, POST, PUT, and DELETE, receiving responses in structured formats such as the JavaScript Object Notation (JSON) [131]. HTTP is the standard protocol used for communication between clients and servers on the web, defining how requests and responses are formatted and transmitted [132]. JSON is a lightweight text-based data interchange format that represents data as human-readable key-value pairs [133]. The construction process involved three stages: API collection, instruction generation, and solution path annotation using a depth-first search-based decision tree algorithm to enhance reasoning capabilities. The resulting model, ToolLLaMA, demonstrated the ability to execute complex instructions and generalize to unseen APIs, exhibiting performance comparable to ChatGPT. ToolLLM also introduced ToolEval, an automatic evaluator for assessing tool-use capabilities.

Additionally, the Berkeley Function Calling Leaderboard (BFCL) provides a benchmark for evaluating LLMs' function-calling and tool-use capabilities across real-world scenarios [134]. BFCL assesses both serial and parallel function calls, supports multiple programming languages, and uses an abstract syntax tree (AST)-based evaluation method that scales to thousands of func-

tions. An abstract syntax tree is a hierarchical, tree-based representation of a program's syntactic structure in which each node denotes a language construct and non-essential syntactic elements are omitted [135]. The benchmark combines expert-curated and user-contributed functions with associated prompts, and measures a model's ability to abstain from unnecessary calls and reason in stateful, multi-step agentic interactions. Evaluations across a wide range of LLMs show that while flagship models perform well on single-turn function calls, memory management, and long-horizon reasoning remain challenging.

A recent survey formally defined the function calling paradigm as the mechanism through which LLMs "analyze user inquiries and engage with defined functions, thereby facilitating precise responses through connections to external sources, including databases, programming interfaces, and live data streams" [136]. Unlike Toolformer, which required fine-tuning the model on filtered API call examples, function calling provides the model with structured function schemas at inference time, typically in JSON format, describing the function name, its purpose, and the expected parameters with their types. The model then determines whether a function is relevant to the user's query and, if so, outputs a structured object containing the function name and extracted arguments rather than a natural language response. The host application executes the function against the external system and returns the result to the model, which uses it to compose its final response. The survey describes this process as comprising three essential phases. Preparation encompasses query analysis and function identification. Execution evaluates whether a function call is necessary, extracts relevant parameters, and oversees the operation. Processing concentrates on analyzing outcomes and crafting appropriate responses.

Function calling defines how a model emits a structured invocation, but every invocation ultimately lands on an interface that the target system exposes. Agents operate such systems through three recurring interface types, the first being software development kits (SDKs).

2.2.4.1 Software Development Kits

An SDK is a packaged collection of libraries, documentation, and code examples through which developers operate a platform or service programmatically from a general-purpose programming language [38]. Its functional surface is an API. For a remote platform, the SDK typically wraps the platform's REST API in bindings for a particular programming language, handling authentication, serialization, pagination, and retries so that platform

operations appear as ordinary function calls.

The quality of an SDK therefore rests on the quality of its API [38]. A good API is characterized as easy to learn, easy to use even without documentation, hard to misuse, and easy to read and maintain in the code that uses it, and a public API can never be changed without breaking its consumers [38]. Subsequent guidance framed the API as the user interface that a software component presents to programmers, argued that design mistakes multiply across every caller and every program built on top of the interface, and recommended designing APIs from the perspective of the caller rather than around the convenience of the implementation [137].

Apart from SDKs, the shell Bourne Again SHell (BASH) has become an increasingly popular type of tool for software and coding agents that need to interact directly with the local operating system and any command line interface (CLI)-based tool installed on the host machine.

2.2.4.2 Command Line Interfaces

A CLI is a text-based interface through which users or programs issue commands as typed strings and receive text output in return. The concept can be traced at least as far back as early time-sharing systems, which allow multiple users to interact with a single computer concurrently by rapidly switching between their tasks. One example is the Compatible Time-Sharing System developed at Massachusetts Institute of Technology (MIT) in the early 1960s, which provided users with an interactive command interpreter for managing files and executing programs from a terminal [138]. The established synonym CLI, emphasizing the user-facing aspect rather than the underlying interpreter, appeared as early as 1983 in research on user-defined interfaces for electronic mail systems [139].

Unix, a general-purpose, multi-user, interactive operating system developed in the early 1970s, played a pivotal role in shaping the modern CLI [140]. It introduced foundational abstractions such as a hierarchical file system, uniform treatment of files and devices through a common input/output interface, and process control primitives that remain central to modern operating systems. The primary means of user interaction with Unix was the Shell, a CLI that reads lines of text typed by the user and interprets them as requests to execute programs. The Shell runs as an ordinary user-level process rather than as a privileged part of the operating system kernel, a design decision that established the enduring separation between the operating system and its command interface.

The Bourne Shell, introduced by Stephen Bourne as a successor to the original Unix shell, unified interactive command execution with a general-purpose programming language, establishing the shell not merely as a CLI but as a scripting environment [141]. The Bourne Shell reads and executes commands either interactively from a terminal or from stored files known as shell procedures. Core features include input and output redirection, which allows commands to read from and write to files rather than the terminal. Pipelines connect the standard output of one command to the standard input of another using the pipe operator. File name generation through pattern matching with wildcard characters and control-flow primitives enable the construction of complex command procedures.

The behavior of the Bourne Shell was later formalized by the Institute of Electrical and Electronics Engineers (IEEE) as part of the Portable Operating System Interface (POSIX) Shell and Utilities specification, a set of open system standards based on Unix that defines the command language, built-in utilities, and interactive facilities a conformant shell must provide [142]. BASH is an implementation of this standard that extends the original Bourne Shell with additional features such as command-line editing, command history, and job control [143]. It can be used interactively, accepting input typed from a keyboard, or non-interactively, executing commands read from a file or a string. It supports both synchronous execution, where the shell waits for a command to complete before accepting further input, and asynchronous execution, where commands run in parallel with the shell.

An early exploration of artificial intelligence (AI) agents operating CLIs was Project Command Line Artificial Intelligence (CLAI) [144]. CLAI provided a heavily instrumented version of BASH, enabling new end-user interaction patterns such as natural language translation of command-line tasks to their correct BASH syntax. It was open-sourced at the 34th Association for the Advancement of Artificial Intelligence (AAAI) Conference on Artificial Intelligence and led to the Natural Language Command to Command (NLC2CMD) competition at the Conference on Neural Information Processing Systems (NeurIPS) 2020, which challenged participants to translate natural language descriptions of command-line tasks into valid BASH commands [145]. The competition revealed several open challenges, including the difficulty of handling underdetermined invocations where natural language descriptions map to multiple valid command translations, the need for session-level evaluation rather than isolated single-command accuracy, and security concerns around command injection through manipulated training data. Participants also noted that the frequency of

real-world commands follows a heavy-tailed distribution, suggesting that focusing on a restricted but frequently used set of commands may yield more practical accuracy than attempting full coverage.

Over the following years, work has focused on improving the evaluation of natural language to BASH command translation. A study presented a manually verified test dataset of 600 instruction-command pairs and a training dataset of 40,939 pairs, increasing the size of previous datasets by 441% and 135% respectively [146]. The study also introduced a novel functional equivalence heuristic that combines command execution with LLM evaluation of command outputs, determining the functional equivalence of two BASH commands with 95% confidence, a 16% increase over previous heuristics. Evaluation of popular LLMs demonstrated that parsing, in-context learning, in-weight learning, and constrained decoding can improve translation accuracy by up to 32%. However, the accuracy of state of the art (SOTA) LLMs for this task remains low, highlighting that natural language to BASH translation continues to be a challenging problem.

While function calling standardized how individual LLMs interact with external tools, it did not address how to standardize the communication layer between agents and the servers that host these tools, which is the problem addressed by the Model Context Protocol (MCP).

2.2.4.3 Model Context Protocol

In late 2024, Anthropic launched the MCP, a universal open standard for defining, discovering, and invoking external tools for AI applications [147]. Drawing inspiration from the Language Server Protocol, an open standard originally developed by Microsoft that decouples programming language support from code editors by defining a common interface through which any editor can communicate with any language server for features such as auto-completion and error checking [148], MCP introduces several features beyond conventional function calling. The protocol decouples tool implementation from usage, enabling developers to publish and describe external functions dynamically, independent of any single model or agent framework. Dynamic discovery and schema negotiation allow clients to list available tools at runtime, retrieve capabilities, and invoke them in a uniform manner without prior hardcoding or platform-specific adapters. Bidirectional communication channels support both model-to-tool requests and tool-initiated events or notifications back to the host. Access control and capability negotiation are designed as first-class features, providing a foundation for more auditable and

secure AI-to-tool interactions. In doing so, the MCP succeeds where earlier agent communication languages such as Knowledge Interchange Format (KIF) and Knowledge Query and Manipulation Language (KQML) did not. Rather than prescribing a declarative language for inter-agent assertions, it standardizes the narrower and more tractable problem of tool discovery and invocation between an agent and its capability providers.

The MCP is guided by a set of principles that prioritize consistency, simplicity, and long-term sustainability [149, 150]. It favors convergence by enforcing a single well-designed solution rather than multiple competing approaches, and emphasizes composability by relying on a small set of core primitives instead of adding specialized features. Interoperability is preferred over optimization, ensuring the protocol functions across diverse systems with varying capabilities, while stability is prioritized over rapid iteration to avoid irreversible design decisions. The protocol also avoids compensating for temporary model limitations, instead focusing on enduring capabilities, and values demonstrated implementations over theoretical discussion. Pragmatism guides decision-making to balance usability with design purity, and standardization is based on proven patterns rather than speculative innovation, ensuring that extensions serve as experimentation grounds before becoming part of the core specification.

An example of domain-specific MCP adoption is the Isabl AI Agent, developed for the Isabl Platform, a precision oncology system broadly adopted at Memorial Sloan Kettering Cancer Center and external research institutions [151]. The platform manages hundreds of projects encompassing over 105,000 sequencing experiments from approximately 70,000 individuals and more than 4.7 petabytes of analytical data. To reduce the learning curve imposed by the platform's depth and schema complexity, the authors developed an MCP server exposing Isabl tools such as API querying and pipeline execution through a standardized interface, combined with retrieval-augmented generation (RAG) over indexed documentation and a ReAct-style agentic controller. ReAct is an approach that enables LLMs to interleave reasoning traces and task-specific actions, allowing them to plan, update, and execute tasks while interacting with external sources for improved performance, interpretability, and reliability [152]. The agent handles tasks ranging from cohort discovery and multi-step genomic reasoning to automated workflow submission via natural language, demonstrating how MCP can bridge general-purpose LLMs with specialized scientific infrastructure.

A systematic study of MCP defines the full lifecycle of an MCP server as comprising four phases [147]. The first phase is creation, in which developers

define metadata, declare capabilities, and implement the server. The second phase is deployment, in which the server is released to a platform and registered with a host. The third phase is operation, in which the server actively interprets user intent, accesses external resources, invokes tools, and manages sessions. The fourth phase is maintenance, which covers version control, configuration changes, and security auditing [147]. Since its release, MCP has rapidly grown into a foundational architecture for AI-native applications, with thousands of independently developed servers exposing model-accessible interfaces to services such as GitHub, Slack, a workplace messaging platform for team communication and collaboration [153], and Blender, an open-source three-dimensional modeling and animation application [154].

A second study examined the architectural role of the MCP in multi-agent systems (MASs), interpreting it not merely as an integration protocol but as an autonomous architectural layer that formalizes the topology of interaction among hosts, clients, and servers [155]. The study identified four mutually reinforcing principles derived from the protocol's design. The first is strict separation of concerns, which keeps cognitive functions such as planning and synthesis distinct from execution and data access. The second is security and governance by design, which relocates access control and auditability to the server layer. The third is modular composability and interoperability, which enables shared capability providers to serve multiple agents across heterogeneous frameworks. The fourth is declarative context provisioning, which transforms prompts and resources into managed assets that can be discovered, versioned, and reused across workflows. The study also proposed a lifecycle interpretation showing that MCP stabilizes system evolution across creation, operation, and update phases, with a customer support reference architecture demonstrating how narrowly scoped cognitive roles can be coordinated by a host while relying on a shared capability layer. However, the study noted that the broader ecosystem still faces unresolved challenges around centralized oversight, multi-tenant configuration, cross-component authentication, and trusted distribution.

Whether an agent reaches a system through the MCP or the CLI, the interface exposes what can be invoked but not which invocations a given task requires, in what order, or with what conventions. That procedural knowledge is supplied separately, through agent skills examined in the following section.

2.2.5 Agent Skills

In the context of agents, skills are defined as "abstractions over useful and reusable behaviors for the target tasks" [156]. SkillAct proposed a prompting approach in which each skill is represented as a tuple of a name, a set of instructions describing how to execute the skill, and examples of the skill in use extracted from task trajectories. By appending these skill descriptions to an agent's prompt, the agent gains access to reusable behavioral templates that guide its planning and action execution across different tasks. The study demonstrated that skills can also be automatically extracted from few-shot demonstrations by prompting a large language model (LLM) to identify shared behaviors across trajectories, producing skills comparable in quality to those written by human experts. Evaluated on Action Learning From (ALF)World, a text-based simulated household environment built on top of the Action Learning From Realistic Environments and Directives (ALFRED) benchmark for embodied agents completing household tasks in visually grounded settings [157], SkillAct improved success rates from 79.7% to 85.1% with extracted skills and to 88.0% with expert-written skills when using generative pre-trained transformer (GPT)-4-turbo, with analysis showing that skills primarily reduced plan execution errors such as incorrect action syntax rather than planning errors.

This notion of skills as reusable, composable behavioral abstractions encoded in natural language has since been adopted by practical coding agent tools, where skills take the form of Markdown (MD) files containing instructions that extend the agent's capabilities and can be invoked automatically or on-demand. An MD file is a plain text file formatted using MD syntax, which allows text, headings, code blocks, and lists to be easily written and read [158].

Recently, these abstractions have been implemented as agentic coding manifests. Files such as `Claude.MD` provide agents with project context, identity, and operational rules [159]. A `Claude.MD` file is a markdown file placed in a project's root directory that is automatically loaded into the context of Claude Code, Anthropic's coding agent [115], at the start of every session, serving as persistent instructions that shape how the agent understands and interacts with the codebase [160]. More recently, the `AGENTS.MD` format has emerged as an open, standardized alternative, making these practices broadly compatible across different coding agents [161].

A study analyzing 253 such files from 242 repositories found that developers prefer a shallow hierarchical structure, typically using a single primary heading with a moderate number of subsections [159]. Content

is dominated by operational commands, technical implementation notes, and high-level architecture descriptions, with many manifests also including artificial intelligence (AI) role definitions to guide agent behavior. These findings highlight the dual purpose manifests serve in agentic coding workflows. They function both as execution guides for technical tasks and as frameworks for human-AI collaboration. This pattern echoes the elementary operations concept from robotics, where a single structure encapsulated both the agent-independent description needed for coordination and the agent-specific implementation needed for execution.

What tools and skills have in common is that neither is placed in the context window in full. The principle that governs how much of each reaches the model is examined in the following subsection.

2.2.6 Progressive Disclosure

The principle that governs how tools and skills are exposed to an agent is progressive disclosure, and predates agents by decades. It was introduced in the retrospective on the Xerox Star, the first commercial system built around a graphical user interface, as the principle that detail be hidden from users until they ask or need to see it [162]. It responded to the observation that systems overwhelm their users with choices, commands to remember, and output that is largely irrelevant to the task at hand, because they treat every possible action and every generated fact as equally likely to matter. The Xerox Star applied it by hiding properties users are unlikely to change until they indicate that they want to change them, and by displaying property sheets on demand rather than permanently.

Progressive disclosure reached agents in three further steps as of 2025. Anthropic reintroduced the term with agent skills, describing it as the design principle that makes them scalable [163]. A skill is exposed to the model in levels. Its name and description sit in the system prompt and carry just enough information for the model to decide whether the skill is relevant, the full instruction file is loaded only once it is, and bundled supplementary files are read-only when the task requires them, so that the agent loads information as needed rather than holding every skill in its context window. The principle was then carried over to tools. An analysis of code execution with the Model Context Protocol (MCP) observed that tool definitions consume context before any tool is invoked and argued for writing them to files that the agent discovers on demand, which costs no tokens until a definition is read and additionally allows the agent to chain calls in code rather than passing every intermediate

response through the context [164]. The step from tools to interfaces was made explicit for the command line interface (CLI), where it was argued that CLI tools already implement progressive disclosure through `-help`, which reveals commands and their parameters hierarchically on request, so that a CLI attains the context efficiency of skills without a separate framework around it [165].

What the Xerox Star hid from human users to spare their attention is therefore hidden from agents to spare their context window. Progressive disclosure is the mechanism to which the practitioner comparisons reviewed in the related work attribute the token advantage of the CLI over the MCP, and it is the property the interface contrast of this thesis tests, since a CLI discloses its surface through help output as the agent needs it, whereas a software development kit (SDK) requires the agent to know or infer the application programming interface (API) it writes against.

While the preceding sections have established the theoretical foundations of agents, tools, skills, and progressive disclosure, the following section examines how these concepts have been applied and evaluated in existing research.

2.3 Related Work

This section reviews existing research relevant to the present work. It is organized into four subsections: meta-harnesses, machine learning (ML), benchmarks, and systems optimization using agents. The first traces the harness, the code, configuration, and execution logic around a model, and then reviews the competing definitions of the meta-harness, the layer above individual agent harnesses, and fixes the definition this thesis adopts. The second examines ML from two perspectives: agents that automate ML workflows and the platforms on which these agents and workflows are executed. The third surveys benchmarks across four areas: coding agents, data and feature engineering, the command line interface (CLI), and ML agents. The fourth reviews approaches for optimizing systems using agents.

2.3.1 Meta-Harnesses

Before meta-harnesses emerge, harnesses appear first as software rather than as a concept. The `agent-harness` package is published to the Python Package Index in August 2023 as the component through which a benchmarking platform runs its agents [166], a name for a piece of plumbing rather than for a category. Scholarly use follows in 2025, when a benchmark paper reports implementing an agentic harness that equips large language models (LLMs)

with tools sufficient to produce accurate responses [167], applying the word to the scaffolding around the model rather than to the model itself.

From there the term reaches practitioners. Mitchell Hashimoto, the founder of HashiCorp and the author of the infrastructure as code tools Terraform and Vagrant, which made the declarative provisioning of infrastructure standard practice, describes harness engineering in early 2026 as the practice of answering every agent mistake by engineering the environment so that the mistake cannot recur, either by writing the correction into an agentic coding manifest or by programming a tool that verifies the work [168]. Open Artificial Intelligence (OpenAI) reports the same practice at production scale in the same period, describing an agent-first codebase built around Codex [169]. What these accounts leave implicit, LangChain makes explicit by decomposing the term. In practice, an agent is a model plus a harness, and the harness comprises every piece of code, configuration, and execution logic that is not the model itself, namely the system prompt, the tools, skills, and Model Context Protocol (MCP) servers exposed to the model together with their descriptions, the bundled infrastructure of file system, sandbox, and browser, the orchestration logic that spawns subagents and routes between models, and the hooks that enforce deterministic steps such as compaction and lint checks [170].

Extending the concept of harnesses, the term meta-harness emerges in 2026 for the layer above individual agent harnesses. Three definitions circulate, one from industry, one from research, and one from open source tooling. Databricks introduced Omnigent as a meta-harness for combining, controlling, and sharing agents, a positioning its chief technology officer has since presented publicly [171, 172]. In this definition, a meta-harness is a runtime layer above agent harnesses with a fixed architecture. A runner wraps any agent, such as Claude Code, Codex, or a custom agent, in a sandboxed session with a uniform application programming interface (API), and a server provides policies and sharing and exposes every session over the terminal, the app, and web APIs. Sessions, policies, and skills stay with the operator regardless of which agent or model is running, and stateful policies track agent actions and enforce guardrails such as cost budgets and permissions.

Research at Stanford defines the term through optimization rather than operation [173]. Their meta-harness is an outer loop system that searches over harness code, where the harness is the code that determines what information to store, retrieve, and present to the model. An agentic proposer, itself a coding agent, reads the source code, scores, and execution traces of all prior candidates and proposes targeted edits, and the authors motivate the search by showing that changing only the harness around a fixed model can produce

a sixfold performance gap on the same benchmark. A meta-harness in this sense is an optimizer that treats the harness as a searchable program.

A third definition comes from the CrewHaus whitepaper, which describes a meta-harness as a compiler and control plane that accepts one high level specification and emits a concrete runtime, with the recurring harness primitives as its intermediate representation and the runtime architectures as its backends [174]. A meta-harness in this sense is a build tool that amortizes harness engineering across deployment shapes.

The three definitions share the view that the gain above individual harnesses lies in a layer that treats agents and their harnesses as interchangeable components, and they differ in whether that layer operates agents, optimizes them, or compiles them. This thesis adopts the Databricks definition, since its subject is the operation of machine learning (ML) platforms. The next subsection turns from the layer that operates agents to the ML agents and platforms themselves.

2.3.2 Machine Learning

This subsection reviews the intersection of agents and platforms in the context of machine learning (ML). It first examines existing approaches to specializing software agents for automating ML workflows, and then reviews existing surveys of ML platforms.

2.3.2.1 Agents

Various approaches have been proposed to create specialized agents for automating ML workflows, forming a class of systems commonly referred to as ML agents. These agents use large language models (LLMs) to reduce manual effort across stages of the automated machine learning (AutoML) pipeline and vary in their degree of platform integration, tool interface design, and pipeline coverage.

One agent has been proposed that enables natural language interaction throughout the entire ML workflow [175]. The agent implements an end-to-end ML pipeline incorporating automatic data loading and preprocessing, task identification, neural architecture selection, hyperparameter optimization, and training automation. It introduces a data processing approach that uses LLMs to automatically interpret and handle diverse data formats without requiring manual pre-processing, as well as an adaptive hyperparameter optimization strategy that combines LLM knowledge of ML best practices with dynamic performance feedback. Evaluation on ten

datasets spanning classification and regression tasks demonstrates increased performance compared to traditional rule-based AutoML frameworks. While the agent covers the complete AutoML pipeline including data preparation, feature engineering, model generation, and model evaluation, it operates through remote data ingestion rather than integration with an ML platform.

Introspective Monte Carlo Tree Search (I-MCTS) addresses the low-diversity and suboptimal code generation that agents in AutoML often exhibit by introducing an introspective process that iteratively expands tree nodes through analysis of solutions and results from parent and sibling nodes [176]. The approach integrates an LLM-based value model to evaluate each node's solution prior to computational roll-outs, and implements a hybrid rewarding mechanism that transitions from LLM-estimated scores to actual performance scores. Applied to various ML tasks, I-MCTS demonstrates a 4% absolute improvement over strong open-source AutoML agents. While the approach targets the complete AutoML pipeline, it focuses solely on the agent's search and reasoning strategy without considering platform integration or tool interfaces such as a software development kit (SDK), a command line interface (CLI), or the Model Context Protocol (MCP).

Within data lake systems, a second line of work integrates AutoML frameworks and LLMs into the Semantic Data Reservoir for Heterogeneous Datasets (SEDAR) platform [177]. SEDAR is a semantic data lake that integrates data ingestion, storage, processing, governance, and semantic enrichment to support data integration, linking, profiling, and ML workflows [178]. The work introduces a metadata model for capturing data analytics processes, a Python package wrapping existing AutoML libraries, and a module utilizing LLMs to automate ML tasks. A comparative evaluation indicates that AutoML simplifies pipeline creation but limits user control and lacks robust data preprocessing support, while LLMs can automate individual tasks such as code generation but struggle to orchestrate complete workflows effectively. The authors propose a vision integrating multi-agent frameworks with knowledge graphs that capture historical experience from previous ML experiments. This approach addresses data preparation, model generation, and model evaluation, but does not employ an SDK, a CLI, or the MCP as tool interfaces, instead relying on an integrated agent architecture within the data lake system.

A third study transforms a conventional application programming interface (API)-based analytics platform into a set of MCP-based tools accessible by agents, establishing a process to facilitate communication between data analysts, agents, and the platform [179]. The resulting chat-based interface allows analysts to query and execute analytical workflows using natural

language. The framework is instantiated with open-source models and achieves a 7.2% mean overall score increase compared to baselines, with complementary user study data demonstrating superior task efficiency and higher user preference scores relative to a traditional form-based interface. While this approach employs the MCP as its tool interface, it is limited to the data preparation stage of the AutoML pipeline and does not address feature engineering, model generation, or model evaluation.

AutoMLPipeline is a closed-loop code generation framework that integrates retrieval-augmented generation (RAG) with reinforcement learning (RL) feedback mechanisms to produce deployment-ready pipeline code for cloud-native ML platforms [180]. The framework uses a knowledge base constructed from successful pipeline execution logs to guide generative pre-trained transformer (GPT)-4 in generating code that adheres to platform-specific constraints, and introduces a pre-validation agent that employs simulated execution environments to predict resource consumption and detect dependency conflicts before deployment. Evaluation on the CodeSearchNetwork (net) dataset augmented with Azure ML pipeline specifications demonstrates a 43.7% improvement in first-submission success rate and a 31.2% reduction in resource over-provisioning compared to vanilla GPT-4 baselines. This approach is the closest to the present work in that it generates executable code targeting the cloud platform Microsoft Azure, covering data preparation, model generation, and model evaluation. However, the framework does not address feature engineering, and it relies solely on code generation without employing external tools such as an SDK, a CLI, or the MCP to interact with the platform at runtime.

Having reviewed agents that automate ML workflows, the following section turns to the platforms on which such workflows are executed.

2.3.2.2 Platforms

Beyond the Hopsworks Feature Store, numerous other data science and ML platforms exist [181]. Gartner, an American research and advisory firm, classifies vendors into four categories as displayed in Figure 2.4: leaders, challengers, visionaries and niche players [182].

Leaders have mature platform strategies that use generative artificial intelligence (GenAI) and agents, and innovate at a pace that outpaces competing vendors [181]. Eight platforms occupy this category, including those from Databricks, Google, Microsoft and Amazon Web Services (AWS). Challengers possess the operational capacity to serve broad enterprise needs but

are constrained by limited geographic or industry focus. Alibaba Cloud is the sole entrant in this category. Visionaries offer differentiated, forward-looking solutions but lack sufficient recognition for end-to-end adoption, and include five platforms from companies such as Snowflake, Statistical Analysis System (SAS) and Cloudera. Niche Players, such as Alteryx and MathWorks, focus on specific industries or user groups but have limited broader market traction.

Figure 2.4: Gartner Magic Quadrant for Data Science and Machine Learning Platforms [181]

While agents feature prominently in the evaluation, the interfaces through which coding agents access these platforms receive little attention. CLIs and SDKs are not assessed as agent-facing interfaces at all. Neither the availability of an SDK, a CLI, an MCP server, or ML platform agents, nor the extent to which these interfaces and agents cover feature, training, and inference tasks, constitutes an evaluation criterion. Consequently, no systematic assessment exists of which ML platforms a coding agent could operate today, through which interfaces and agents, and across which machine learning operations (MLOps) and AutoML tasks. Establishing this landscape is a prerequisite for benchmarking coding agents on ML platforms and motivates RQ1.

With the initial platform landscape established, the following subsection reviews how agents operating on such platforms could be evaluated, surveying existing benchmarks.

2.3.3 Benchmarks

The following subsections review existing benchmarks across four areas relevant to the research questions (RQs): coding agents, data and feature engineering, the command line interface (CLI), and machine learning (ML) agents.

2.3.3.1 Coding Agents

Existing benchmarks for coding agents span a broad range of capabilities, including safety, data science, autonomous task execution, repository-level code generation, automated program repair, communication skills, and collaborative coding.

In the domain of safety, RedCode provides an evaluation platform for assessing the security of code agents across unsafe code execution and generation [183]. It covers 25 types of critical vulnerabilities spanning websites, file systems, and operating systems, and evaluates agents across multiple input formats including Python, Bourne Again SHell (BASH), and natural language descriptions.

Several benchmarks target code generation in data-intensive and repository-level contexts. Data Analysis (DA)-Code evaluates large language models (LLMs) on agent-based data science tasks grounded in real and diverse data, requiring complex data wrangling and analytics using Python and the structured query language (SQL) [58]. Application World (AppWorld) provides an execution environment of nine applications operable via 457 application programming interfaces (APIs), with 750 intelligent agent tasks requiring rich and interactive code generation [184]. CodeAgent introduces a framework with five programming tools for repository-level code generation, evaluated on the CodeAgentBenchmark (bench) dataset and HumanEval [185]. GitTaskBench assesses coding agents on 54 realistic tasks across seven modalities and seven domains, pairing repositories with automated evaluation harnesses and introducing an alpha-value metric to quantify economic benefit [186]. ProjectEval simulates user interaction for project-level code generation evaluation across three input levels [187].

Automated program repair represents another active area. A survey traces the evolution from template and constraint-based approaches

to LLM-powered systems, comparing retrieval-augmented, agent-based, and agentless paradigms across benchmarks such as software engineering (SWE)-bench, CodeAgent-Bench, and CodeRetrieval-augmented generation (RAG)-Bench [118]. Model Diagram Code (MDC)Agent addresses model diagram-to-code generation through a collaborative multi-agent framework for parsing, generating, and verifying code from model diagrams across 16 research domains [188].

Further benchmarks evaluate broader coding agent capabilities. Massive Multitask Agent Understanding (MMAU) provides offline evaluation across five domains including tool use, data science and ML coding, and contest-level programming, covering capabilities such as understanding, reasoning, planning, problem-solving, and self-correction [189]. Human Evaluation Communication (HumanEvalComm) assesses the communication skills of LLMs for code generation, evaluating their ability to ask clarifying questions when problem descriptions contain inconsistencies, ambiguities, or incompleteness [190]. CooperBench evaluates collaborative coding through 600 tasks where two coding agents must coordinate on independently implementable but potentially conflicting features, revealing a 30% average drop in success when agents work together versus individually [191]. Code Semantic Question Answering (CoSQA)+ introduces an automated pipeline for semantic code search evaluation with test-driven agent annotation [192]. BenchAgents proposes a framework that uses LLMs to automate benchmark creation for complex capabilities through planning, generation, data verification, and evaluation agents [193].

While MMAU includes data science and ML coding as one of its evaluation domains, it assesses general coding competence on ML tasks rather than evaluating coding agents that operate an ML platform through its interfaces. None of the reviewed benchmarks evaluates coding agents performing feature, training, and inference tasks on an ML platform, jointly measure accuracy, cost, and the number of turns required, or compare agents of different origins and model tiers under the presence and absence of skills.

Having outlined the broader benchmarking landscape for coding agents, the following section explores benchmarks that specifically target data and feature engineering.

2.3.3.2 Data and Feature Engineering

Beyond general coding tasks, several benchmarks target the data-centric work that precedes and surrounds model training. DSEvaluation (eval) broadens

the evaluation scope from isolated code generation to the full lifecycle of data science agents [194]. Rather than assessing only whether generated code is correct, the paradigm continuously monitors the query, the retrieved context from a stateful runtime session, the generated code, the execution result, and the resulting session state through modular validators, including an intactness validator that fails an agent for unintentionally modifying existing variables. To reduce annotation effort, benchmarks are constructed through an LLM bootstrapping process with humans in the loop, expressed in a dedicated annotation language, yielding four benchmarks with 825 problems drawn from tutorial exercises, question and answer pages, and platforms for coding and data science competitions. Six agent frameworks were evaluated, revealing that presentation errors and intactness violations are common failure modes, that context selection, particularly code history, strongly influences performance, and that multi-agent frameworks incur significantly higher costs without demonstrating clear advantages over single-agent designs.

Data Science Benchmark (DataSciBench) extends this line of work toward complex, multi-step data science prompts with uncertain ground truth [195]. The benchmark comprises 222 prompts spanning six task types, from data cleaning and exploration to visualization, predictive modeling, mining, and report generation, with ground truth generated through a semi-automated pipeline combining LLM self-consistency and human verification. Its Intention-Function-Code framework maps each prompt to intent-specific task types, verifiable evaluation functions, and programmatic validation code, producing 519 test cases evaluated through 25 aggregate metrics. Across 26 evaluated models, API-based models outperform open-source counterparts, and the results expose persistent weaknesses in following fine-grained instructions, calling appropriate tools, executing accurate plans, and exporting required outputs. Even models proficient in reasoning tasks fail on data science workflows for these reasons. Of the 26 models, however, 25 are raw LLMs invoked through a fixed agentic scaffold, with only a single dedicated agentic model included in the comparison.

While these benchmarks cover broad data science workflows, Feature Engineering (FeatEng) concentrates on the single most knowledge-intensive stage within them and evaluates the ability of LLMs to perform feature engineering on tabular data [196]. Given a textual description of one of 103 manually curated Kaggle datasets, including metadata and semantic explanations of its columns, the model must generate a Python function that transforms the raw data into a more predictive feature set. Quality is measured objectively as the percentage error reduction of an eXtreme Gradient Boosting

(XGBoost) model trained on the transformed data relative to a baseline trained on the untransformed data. Success on the benchmark requires the integration of multiple skills, including data interpretation, multi-step reasoning, code generation, and, most distinctively, the application of world and domain knowledge, such as enriching a dataset by mapping countries to their approximate gross domestic product per capita. Analysis of generated features confirms that strong domain knowledge based features have a statistically significant positive impact on scores, and the benchmark exhibits a strong rank correlation of 0.878 with Chatbot Arena ratings at a fraction of the evaluation cost. The best-performing model at the time of publication achieved an average error reduction of roughly 11%, only slightly above a constrained automated machine learning (AutoML) baseline. Notably, models are evaluated in a single pass by design, and the authors acknowledge that the protocol does not assess iterative refinement, deferring agentic evaluation to future work.

Across all three benchmarks, evaluation takes place in generic Python environments built on libraries such as Pandas, Scientific Python Toolkit (scikit)-learn, and XGBoost. None of them evaluates data or feature engineering against an actual ML platform, its feature store, or its native tooling. Agents, meanwhile, are covered only partially, as FeatEng deliberately excludes agentic behavior, DataSciBench evaluates predominantly raw models, and only DSEval places agent frameworks at the center of its design. FeatEng in particular acknowledges this limitation explicitly and defers agentic evaluation to future work. This thesis takes up that direction and extends it in two ways: from single-pass LLMs to coding agents that iteratively refine their solutions, and from local Python environments to an ML platform operated across the full range of machine learning operations (MLOps) and AutoML tasks.

However, operating a platform requires an agent-facing interface, and the following section therefore reviews benchmarks and evidence on the CLI in this role.

2.3.3.3 Command Line Interface

Benchmarking of LLM-based agents operating CLIs remains limited in the academic literature. Terminal-Bench 2.0 is the only peer-reviewed benchmark specifically designed to evaluate agents on tasks in computer terminal environments [197]. It comprises 89 curated tasks inspired by real workflows, each featuring a unique environment, a human-written solution, and comprehensive tests for verification. Among its tasks, the benchmark includes several ML-related challenges [198]. One task requires

training a convolutional neural network (CNN) on Canadian Institute for Advanced Research (CIFAR)-10, a widely used image classification dataset consisting of 60,000 color images across ten classes [199], using the Caffe framework, an early open-source deep learning (DL) library developed at the Berkeley Vision and Learning Center [200].

Another task involves reconstructing and fine-tuning a PyTorch model from a state dictionary [201], a serialized representation of a model's learned parameters. A third task requires implementing pipeline parallel training for the Large Language Model Meta Artificial Intelligence (LLaMA) model, a technique that partitions model layers across multiple devices to enable training of large models that exceed the memory capacity of a single device. A fourth task involves training a FastText model, a lightweight text classification and word representation library developed by Facebook Research [202], on review data. A fifth task requires building a command-line tool for inference on Modified National Institute of Standards and Technology (MNIST), a benchmark dataset of handwritten digit images commonly used for evaluating image classification models [203]. These tasks partially address the model generation and model evaluation stages of the AutoML pipeline. Evaluation of frontier models and agents shows that none exceeds 65% on the benchmark, with error analysis identifying areas for model and agent improvement. However, Terminal-Bench 2.0 does not systematically cover the full AutoML pipeline, including data preparation and feature engineering, does not evaluate agents operating an ML platform, and does not compare CLI-based interaction against a platform's native software development kit (SDK).

In contrast to the scarcity of academic work, considerable practitioner effort has been devoted to evaluating CLIs as tool interfaces for coding agents, typically in direct comparison with the Model Context Protocol (MCP). This discourse began in August 2025 and has continued since, primarily through technical blog posts, GitHub discussions, and threads on platforms such as Hacker News, and is reviewed here because it constitutes the only available evidence on how CLI-based interfaces serve coding agents.*

An early pair of posts evaluated equivalent MCP and CLI versions of a terminal-control tool across 120 runs, finding equivalent success rates and similar costs between the two variants [204]. A follow-up argued that established MCP servers such as Playwright MCP consume 13,700 to 18,000 tokens of context across 21 to 26 exposed tools, whereas a 225-token file documenting four equivalent CLI scripts achieved the same functionality,

*Where available, the corresponding Hacker News discussion thread is included in the bibliography entry of the primary source.

a roughly 60-fold reduction [205]. Together, the posts converge on the position that the choice between the two interfaces is largely determined by tool design rather than the protocol itself.

A separate analysis quantified the context overhead of the MCP relative to a CLI-based equivalent under a typical configuration of six MCP servers exposing 84 tools in total [206]. The MCP integration was reported to load the full JavaScript Object Notation (JSON) schema for every tool at session start, consuming approximately 15,540 tokens, whereas an equivalent CLI-based setup loaded only a lightweight listing of tool names and locations, consuming approximately 300 tokens, with tool definitions discovered on-demand. Across configurations ranging from a single tool call to 100 tool calls per session, the CLI-based setup was reported to use roughly 92 to 98 percent fewer tokens than the MCP, and to remain cheaper than MCP-based tool search mechanisms designed to mitigate the schema-loading overhead.

A GitHub-focused benchmark compared the MCP, the CLI, and a CLI variant augmented with skills on five read-only tasks, holding the model and prompts constant across 75 runs [207]. Token usage was reported to be 4 to 32 times higher for the MCP than for the CLI, attributed primarily to schema injection. The evaluated server exposed 43 tools, all of whose schemas were loaded into the conversation context regardless of which tools were used. Reliability also differed substantially, with the CLI variants completing all 25 runs successfully and the MCP variant failing in 7 of 25 runs due to remote connection timeouts. Augmenting the CLI with an 800-token skill document reduced tool calls and latency by roughly one third compared to the unaugmented CLI. The post nevertheless argued that the MCP's overhead is justified in multi-user, multi-tenant deployments where per-user authorization, tenant isolation, and structured audit trails are required.

A complementary analysis framed the choice between the two interfaces as a function of where in the software delivery lifecycle the agent operates [208]. CLI tools were argued to fit the inner loop of rapid local iteration because of their low context-window overhead, their familiarity to models from training data, and their composability through standard shell mechanisms, whereas the MCP was argued to fit the outer loop of shared infrastructure because of its centralized authentication, structured responses, and session state, with production setups typically requiring both interfaces rather than one.

A further critique pushed back on the prevailing narrative that the CLI uniformly outperforms the MCP, arguing that the comparison shifts substantially depending on whether the CLI tools in question are well-known utilities or bespoke implementations [209]. Standard tools such as `git`, `curl`, `jq`,

or cloud provider CLIs were reported to be highly token-efficient because models have encountered them extensively in training data and can invoke them without explicit instruction. Bespoke CLIs, by contrast, were argued to require descriptions in agentic coding manifests or progressive disclosure of `-help` output to discover commands and parameters, characterized as "the MCP schema, just without any structure" [209]. The post further distinguished between MCP servers running locally over standard input and output and remote MCP servers running over streamable Hypertext Transfer Protocol (HTTP), arguing that only the former is meaningfully comparable to a CLI, while the latter offers organizational benefits such as centralized authentication, telemetry, observability, and dynamic delivery of prompts and resources that are difficult to reproduce with locally distributed CLI tools.

Taken together, the practitioner comparisons cover terminal control, browser automation and debugging, tool discovery overhead, GitHub workflows, the distinction between trained and bespoke tools, and the role of interface choice across stages of the development lifecycle. Their findings converge. CLI-based interfaces consume substantially fewer tokens because tool definitions are discovered on-demand rather than loaded into the context upfront, a pattern corroborated by a browser-debugging comparison reporting 33 percent higher token efficiency for the CLI [210] and by an industry report finding three MCP servers consuming approximately 55,000 tokens before any user message was processed [211]. CLI-based interfaces further achieve equal or higher reliability and benefit from the familiarity of models with standard utilities from training data, whereas the advantages of the MCP concentrate in remote, multi-user deployments requiring centralized authentication, tenant isolation, and audit trails. A single experiment additionally suggests that augmenting a CLI with skills reduces tool calls and latency [207].

This asymmetry between practitioner activity and academic coverage demonstrates that CLI-based interface benchmarking for coding agents is academically underdeveloped. None of the practitioner comparisons is peer-reviewed, and all of them target general software development rather than ML workflows. Combined with Terminal-Bench 2.0, which covers only isolated ML tasks, no benchmark, academic or practitioner, evaluates coding agents operating an ML platform through a CLI across MLOps and AutoML tasks, compares CLI-based interaction against the platform's native SDK, jointly measures accuracy, cost, and turns, or isolates the contribution of skills.

The following section turns to benchmarks that evaluate agents on ML tasks.

2.3.3.4 Machine Learning Agents

Several benchmarks have been proposed to evaluate agents specifically on ML tasks, ranging from experimentation and engineering to code generation for ML algorithms.

MLAgentBench introduces a suite of 13 tasks designed to assess whether LLM-driven agents can perform ML experimentation effectively [212]. Tasks range from improving model performance on CIFAR-10 to recent research problems such as Baby Language Model (BabyLM), a shared task focused on training language models on developmentally plausible amounts of data [213]. Agents can perform actions such as reading and writing files, executing code, and inspecting outputs, following the ReAct framework. Evaluation across several frontier models finds that a Claude v3 Opus agent achieves the highest success rate at 37.5%, though performance varies considerably, spanning from 100% on well-established datasets to 0% on recent Kaggle challenges. The benchmark identifies long-term planning and hallucination reduction as key challenges. While MLAgentBench evaluates agents across multiple stages of ML experimentation, it does not assess agents operating an ML platform through tool interfaces such as an SDK, a CLI, or the MCP.

MLGym provides both a framework and the benchmark MLGym-Bench for evaluating and developing LLM agents on artificial intelligence (AI) research tasks [214]. MLGym-Bench comprises 13 diverse, open-ended research tasks spanning computer vision, natural language processing, reinforcement learning (RL), and game theory. Solving these tasks requires a broad range of research skills, including generating hypotheses, creating and processing data, implementing ML methods, training models, running experiments, analyzing results, and iterating to improve performance. By framing these tasks as a Gym environment, MLGym additionally enables research on RL algorithms for training such agents. Evaluation of several frontier models, including Claude 3.5 Sonnet, generative pre-trained transformer (GPT)-4o, o1-preview, Llama-3.1 405B, and Gemini-1.5 Pro, shows that current models can improve on given baselines, typically by finding better hyperparameters, but do not generate novel hypotheses, algorithms, or architectures. While MLGym broadens the scope of ML agent evaluation to open-ended research and provides an extensible framework for developing new learning algorithms, it does not evaluate agents interacting with an ML platform through standardized tool interfaces.

Building on MLAgentBench, machine learning engineering (MLE)-bench curates 75 ML engineering competitions from Kaggle to measure how well

AI agents perform at real-world ML engineering tasks such as training models, preparing datasets, and running experiments [215]. Human baselines are established using Kaggle's publicly available leaderboards. The best-performing setup, Open Artificial Intelligence (OpenAI)'s o1-preview with Artificial Intelligence-Driven Exploration in the Space of Code (AIDE), achieves at least the level of a Kaggle bronze medal in 16.9% of competitions. AIDE is an ML engineering agent that frames ML engineering as a code optimization problem, using tree search over potential solutions to iteratively refine and reuse promising approaches [216]. The benchmark additionally investigates resource scaling for AI agents and the impact of pre-training contamination. MLE-bench provides a broader and more challenging evaluation than MLAgentBench, but similarly does not evaluate agents interacting with an ML platform through standardized tool interfaces.

MLAlgo-Bench, which references MLE-bench, introduces two tasks focused on ML code generation [217]. The first requires generating code for ML algorithms including both traditional and deep learning-based methods. The second provides human-authored solution sketches and requires writing ML code for practical tasks in Kaggle competitions. Evaluation using an automatic framework with metrics such as task pass rate, relative performance, and time overhead shows that the top-performing model at the time of publication, Claude 3.5 Sonnet, achieves a 48.8% completion rate on algorithm implementation and 21.6% on Kaggle competitions. While MLAlgo-Bench evaluates model generation and model evaluation, it does not address data preparation or feature engineering as autonomous stages, nor does it assess agents operating ML platforms through tool interfaces.

In summary, existing ML agent benchmarks evaluate agents on experimentation, engineering, and code generation tasks, but none assesses agents performing feature, training, and inference tasks on an ML platform through its native interfaces. No benchmark jointly measures accuracy, cost, and turns for such workflows, compares coding agents of different origins and model tiers, or isolates the effect of skills on agent performance. The following subsection turns to approaches for optimizing systems using agents.

2.3.4 Systems Optimization Using Agents

Beyond automating machine learning (ML) workflows, a growing body of work employs agents to optimize the systems on which such workloads run.

A first line of work applies frontier large language model (LLM) agents to join order optimization, a classic database problem in which the number

of possible execution plans grows exponentially with the number of joined tables and where traditional optimizers frequently select poor plans due to misestimated cardinalities [218]. Rather than placing the LLM in the latency-critical path of the query optimizer, the approach frames the agent as an offline experimenter that emulates the iterative tuning process of a human database administrator. The agent is equipped with a single tool that executes a candidate join order and returns its runtime alongside the size of each computed subplan, and the validity of proposed orderings is guaranteed through structured model outputs constrained by a grammar admitting only valid reorderings. Evaluated on the Join Order Benchmark, a suite of 113 queries designed to be difficult to optimize, the agent improved upon the Databricks optimizer in 80% of cases, reducing query latency by a geometric mean factor of 1.288 and lowering P90 latency by 41%, thereby outperforming perfect cardinality estimates, smaller models, and a recent offline optimizer. The approach constitutes an anytime algorithm in which larger iteration budgets translate into further performance gains. While this work demonstrates that agents can autonomously improve a data system through iterative experimentation and feedback, the optimization target is the execution plan of individual queries rather than the interfaces and skills through which agents themselves operate a platform, and the setting is a data platform rather than an ML platform.

database (DB)PlanBenchmark (bench) extends this direction from individual join orders to general physical plan optimization [219]. The harness targets the DataFusion query engine and exposes physical query plans through a compact serialized representation, to which an LLM proposes localized edits applied as JavaScript Object Notation (JSON) patches. A test-time optimization workflow combines semantic reasoning over column semantics, value distributions, and domain context, which classical statistics-based optimizers cannot exploit, with an evolutionary search that refines candidate plans across iterations. The evaluation focuses on Online Analytical Processing (OLAP) queries. OLAP denotes a class of database workloads oriented toward analytical processing, in which complex, read-heavy queries aggregate large volumes of data for reporting and decision support, in contrast to transactional workloads that consist of frequent, small read and write operations [220]. Because such queries are executed repeatedly, small efficiency gains translate into substantial cumulative savings, and the evaluation specifically targets join reordering and join-side selection, where cardinality-estimation errors compound multiplicatively. Median speedups reach 1.10 to 1.12 times on the Transaction Processing Performance Council

ad hoc decision support benchmark (TPC-H) and 1.05 to 1.07 times on the Transaction Processing Performance Council decision support benchmark (TPC-DS), two standard suites of analytical structured query language (SQL) queries over synthetic databases, with individual queries accelerated by up to 4.78 times, and optimizations discovered at small scale factors are shown to transfer to larger ones, supporting a low-cost small-to-large workflow. As with agentic join ordering, however, the object of optimization is the query plan itself. The approach neither optimizes the agent-facing interfaces of a platform nor addresses ML workloads.

Closest to the present work is a study on data-centric optimization of lakehouse agents, which optimizes the skills and agentic coding manifests through which coding agents operate Bauplan, an agent-first branching lakehouse [221]. The authors observe that when agents operate production data systems, the textual artifacts surrounding them become part of the system control surface and must be optimized and evaluated as such. Their central insight is that a branching lakehouse turns agent evaluation from an output-matching into a state-verification problem. Because the platform is headless, with all interactions wrapped by a command line interface (CLI) and a typed Python software development kit (SDK), and because Git-like data primitives map every write to an immutable commit, agent-generated pipeline code induces concrete, inspectable lakehouse changes. The optimization pipeline generates task-verifier pairs from a taxonomy grounded in anonymized production traces, executes candidate skills in isolated sandboxes, and scores trajectories at three levels: tool traces, lakehouse state, and final response quality. Using the black-box optimizer Genetic-Pareto (GEPA) with Claude Code and claude-sonnet-4-6 as the agentic setup, optimized skills improved held-out reward relative to their manually curated seeds by 12.0% on average across a benchmark of 780 tasks, with gains of up to 28.6%. Notably, skills involved in multi-skill tasks could not be optimized in isolation, which the authors identify as motivation for jointly optimizing skills in future work. While this study establishes that skills are optimizable system parameters and that write-path verification enables reliable scoring, it differs from the present work in two aspects. First, optimization is performed by a custom optimization system that mutates skill text based on reward signals, rather than by a coding agent that itself analyzes and improves the interfaces through which it operates the platform. Second, the study optimizes skills alone while holding the CLI and SDK constant, and does not compare an optimized interface against the platform's unoptimized CLI and native SDK baselines on accuracy, cost, and turns.

Taken together, these studies demonstrate that agents can improve data systems through iterative experimentation, that plan-level optimizations discovered by agents transfer across scales, and that the textual artifacts surrounding agents constitute optimizable system parameters. However, all three operate on data platforms and query engines rather than ML platforms, and none examines whether a coding agent can itself optimize the CLI and skills through which it operates such a platform, nor how the resulting interface compares against unoptimized CLI and native SDK baselines. This gap motivates RQ3 of this thesis.

2.4 Summary

This chapter presented the foundational concepts and related work underpinning this thesis. The progression began with artificial intelligence (AI), broadly defined as algorithms capable of performing tasks that typically require human intelligence, and narrowed to machine learning (ML), its subfield concerned with algorithms that learn to solve tasks from data without explicit programming. ML engineering introduced the practical principles governing effective model development, including the interplay of representation, evaluation, and optimization, the challenge of overfitting, and the structured AI lifecycle spanning design, development, and deployment. Machine learning operations (MLOps) then extended these principles to the operational level, drawing on development and operations (DevOps) practices such as continuous integration and continuous delivery (CI/CD) to enable fast, reproducible, and reliable releases of ML applications. ML platforms were presented as the infrastructure supporting these practices, with the Hopsworks Feature Store illustrating how feature pipelines, training pipelines, and inference pipelines can be organized around a centralized feature store with support for batch, streaming, and request-time computation.

The chapter then examined deep learning (DL), the subset of ML that learns hierarchical representations through multiple successive layers, reducing the need for manual feature engineering. Generative artificial intelligence (GenAI) was introduced as the branch of DL concerned with generating previously unseen synthetic content, tracing the evolution from early probabilistic models such as Restricted Boltzmann Machines (RBMs) and deep belief networks (DBNs) through variational autoencoders (VAEs) and generative adversarial networks (GANs) to the transformer architecture, which replaced recurrent mechanisms with self-attention to enable parallelized training. Large language models (LLMs) were then presented as the class

of transformer-based models capable of executing diverse natural language processing (NLP) operations, with key concepts including tokenization, attention mechanisms, reinforcement learning (RL) from human feedback, in-context learning, the constraints imposed by the context window, and the biases these models encode and propagate from their training data. Code generation was discussed as the most prevalent application of LLMs within software engineering (SWE), evaluated through benchmarks such as SWE-bench and Deep Evaluation (DeepEval), alongside persistent error categories and robustness limitations. Vibe coding was introduced as the emerging development practice that uses these models, characterized by conversational code generation, multimodal expression processing, and context-aware generation patterns. Automated machine learning (AutoML) was then discussed as the automated construction of ML pipelines on a limited computational budget, comprising data preparation, feature engineering, model generation, and model evaluation, with LLM contributions reviewed across all four stages. Finally, sovereign AI was introduced as a nation's capacity to develop, operate, and control AI systems using domestic infrastructure, data resources, workforce expertise, and commercial ecosystems, independent of foreign platforms, with the origin and scale of models and the origin of ML platforms identified as strategic variables motivating the comparative dimension of RQ2.

The discussion then shifted to agents, beginning with intelligent agents defined by the properties of autonomy, social ability, reactivity, and proactiveness, and formalized through the belief-desire-intention (BDI) model of beliefs, desires, and intentions. Distributed artificial intelligence (DAI) and multi-agent system (MAS) extended these concepts to environments where multiple self-interested agents interact, introducing coordination through commitments and conventions, planning through shared and collaborative plans, and reasoning about outcomes through utility functions. Software agents were presented as the practical translation of these theoretical foundations into autonomous programs operating within computational environments, classified into collaborative, interface, reactive, hybrid, and other types. Coding agents were then introduced as specialized software agents, consisting of planning, memory, perception, and action components, and capable of interpreting developer goals, generating code, and interacting with external tools. Context engineering was examined as the discipline of optimizing external context before presenting it to the model, encompassing techniques such as file ordering, chunking strategies, retrieval-augmented generation (RAG), and memory frameworks like Mem0. Tools were then discussed beginning with self-supervised tool learning through Toolformer, progressing

through fine-tuned tool-use models such as Gorilla and ToolLLM, and culminating in the function calling paradigm, before the three interface types through which agents operate external systems were examined. The software development kit (SDK) was presented as the programmatic interface through which a platform is operated from code. The command line interface (CLI) was presented as a complementary tool interaction modality, tracing its origins from early time-sharing systems and the Unix shell through the Bourne Shell and Portable Operating System Interface (POSIX) standardization to Bourne Again SHell (BASH), and reviewing research on natural language to BASH translation. The Model Context Protocol (MCP) was presented as the agent-native interface type, a universal open standard for defining, discovering, and invoking external tools that supports dynamic discovery, bidirectional communication, and access control. Skills were examined as reusable behavioral abstractions over useful behaviors for target tasks, represented as named instructions with examples that guide agent planning and execution, and implemented in practical coding agent tools as Markdown (MD) files and agentic coding manifests that provide agents with project context, and operational rules. Progressive disclosure was traced from its origin in graphical user interface design, where detail is hidden from users until they ask or need to see it, through its reintroduction for skills, its application to tool definitions discovered on demand, and its identification in the `-help` mechanism of the CLI, establishing the principle by which an interface reaches the model in levels rather than in full.

The related work section examined four areas closely aligned with this research: meta-harnesses, ML, benchmarks, and systems optimization using agents. The meta-harnesses subsection traced the harness from a Python package in 2023 through its first scholarly use to the component definition that separates a model from the code, configuration, and execution logic around it, and then reviewed three competing definitions of the layer above it, a runtime layer that wraps any agent in a sandboxed session with a uniform application programming interface (API), an optimizer that searches over harness code, and a compiler that emits runtimes from one specification, and adopted the first, since the meta-harness built for this thesis is such a layer. The ML subsection reviewed both agents and platforms. ML agents were examined as a class of LLM-based systems that automate ML workflows, varying in pipeline coverage, platform integration, and tool interface design. Existing approaches either cover the complete AutoML pipeline without integration into an ML platform, employ tools but are limited to a single interface type or pipeline stage, or integrate within platform

architectures without standardized tool interfaces. The landscape of ML platforms was surveyed through industry analysis, encompassing leaders such as Databricks, Google, Microsoft, and Amazon Web Services (AWS), as well as challengers, visionaries, and niche players. While agents are featured prominently in these analyses, the interfaces through which coding agents can operate the platforms, and how comprehensively these interfaces cover feature, training, and inference tasks, are not assessed.

Benchmarking was reviewed across four areas. Coding agent benchmarks span safety, data science, repository-level code generation, automated program repair, communication skills, and collaborative coding, but none evaluates agents performing ML tasks on ML platforms. Data and feature engineering benchmarks evaluate models and agents in generic Python environments rather than against an ML platform, with Feature Engineering (FeatEng) explicitly deferring agentic evaluation to future work. CLI benchmarks remain limited to Terminal-Bench 2.0 in the academic literature, while extensive practitioner comparisons against the MCP consistently report CLI advantages in token efficiency, reliability, and model familiarity, with MCP advantages concentrated in multi-user, multi-tenant deployments. Yet none of this work, academic or practitioner, covers MLOps and AutoML tasks, evaluates agents operating an ML platform, or compares the CLI against a platform's native SDK. ML agent benchmarks evaluate experimentation, engineering, and code generation tasks but do not assess agents operating ML platforms.

Systems optimization using agents was then reviewed through three representative approaches. The first is agentic join order optimization, in which an offline LLM agent iteratively experiments with execution plans and outperforms a commercial query optimizer. The second is harness-based physical plan optimization, in which an LLM proposes localized plan edits refined through evolutionary search on an open-source query engine. The third is data-centric optimization of the skills and agentic coding manifests through which coding agents operate a branching lakehouse. Together, these studies demonstrate that agents can improve data systems through iterative experimentation, that plan-level optimizations transfer across scales, and that the textual artifacts surrounding agents constitute optimizable system parameters. However, all three target data platforms and query engines rather than ML platforms, and none examines whether a coding agent can itself optimize the interfaces and skills through which it operates such a platform.

Across these areas, three gaps emerge. First, industry analyses evaluate ML platforms on their capabilities but not on their agent-facing interfaces and agents, leaving open which platforms a coding agent could operate

today via an SDK, a CLI, an MCP server, or ML platform agents, and how comprehensively these interfaces cover feature, training, and inference tasks. Second, no benchmark evaluates coding agents operating ML platforms across MLOps and AutoML tasks, and consequently nothing is known about how coding agents of different origins and model tiers compare on such workloads, how their performance varies across platforms of different origins, or how much skills contribute to it, a comparison of particular relevance given the emergence of European models and platforms alongside their American counterparts. Third, while systems optimization research establishes that agents can improve query plans and that skills constitute optimizable system parameters, no study examines whether a coding agent can itself optimize an ML platform's CLI and skills, and how the result compares against the platform's unoptimized CLI and native SDK. This thesis addresses these gaps through MLPlatformAgentBench (MLPAB), a benchmark for coding agents operating ML platforms that measures accuracy, cost, and turns, and by answering the following three research questions (RQs):

- **RQ1:** Which ML platforms exist, and to what extent do their interfaces, namely SDKs, CLIs, and MCP servers as well as ML platform agents, cover MLOps and AutoML tasks when assessed against a systematic set of coverage criteria?
- **RQ2:** How do European and American coding agents, based on frontier-tier and lower-tier model products, compare in operating European and American ML platforms across MLOps and AutoML tasks with and without skills, measured by accuracy, model invocation cost, and turns?
- **RQ3:** To what extent can a coding agent optimize an ML platform's CLI and skills, and how does the optimized interface compare against the platform's unoptimized CLI and native SDK baselines on accuracy, cost, and turns?

The next chapter presents the method by which these three questions are answered.

Chapter 3

Method

Answering the three research questions (RQs) of this thesis requires an instrument that does not yet exist, namely a way to run coding agents against real machine learning (ML) platforms under identical conditions and grade what they actually produce on the platform, not what they claim to have produced in their transcripts. Building that instrument is therefore part of the method itself, alongside the experiments it enables.

The method consists of five phases, benchmark generation, market assessment, comparison, optimization, and evaluation. It begins with the instrument that all five depend on, MLPlatformAgentBench (MLPAB), a benchmark and meta-harness for evaluating coding agents operating ML platforms, whose construction is part of the method. The benchmark's tasks are generated as fresh, seeded instances of the feature, training, and inference pipelines, including large language model (LLM) fine-tuning and serving, complemented by operations tasks and two capstone projects. An agent's work is graded by reading the resulting platform state back through the platform's own data paths rather than by inspecting the agent transcript. The meta-harness executes each task across a grid of conditions spanning the coding agent, the platform, the interface given to the agent, and the presence or absence of skills. The interface is the platform command line interface (CLI), its Python software development kit (SDK), or its Model Context Protocol (MCP) server, of which the committed experiments compare the CLI and the SDK. For every run, the meta-harness records accuracy, cost, and turns, alongside a detailed timing and tool call decomposition. The planning of the experiments builds on the constructed meta-harness.

The market assessment phase answers RQ1 through a structured scoring built on the benchmark. Candidate ML platforms are systematically scored on

whether each of the agent-facing interfaces and agents, the Python SDK, the CLI, the MCP server, and ML platform agents, covers the feature, training, and inference stages that the benchmark operationalizes as well as operations. An ML platform agent is a vendor-built, platform-specific agent that carries out the feature, training, and inference workloads of its own platform, and coverage in that class is credited to such an agent only. The second route by which an agent operates a platform, a meta-harness that hosts general-purpose coding agents in the platform in the sense adopted in the background, is scored as a criterion of its own rather than folded into the vendor-agent class, because the two differ in what they offer an agent. A vendor agent covers stages with its own capabilities, while a meta-harness covers none by itself and instead admits an arbitrary coding agent operating the platform's other interfaces. Agents that a vendor ships but that meet neither definition are recorded separately as other software agents and are not scored. An other software agent is a software agent published for the platform whose scope lies outside the machine learning operations (MLOps) and automated machine learning (AutoML) tasks the benchmark operationalizes, such as a software agent for analytics or office work, or whose role is to suggest and generate rather than to execute, such as an assistant that requires approval for every action. The category exists so that the judgement is documented with its source rather than dropped, and so that the presence of an agent product is not mistaken for coverage of MLOps and AutoML tasks. The assessment also scores hosting support for MCP servers, documented agent deployments, and a published set of skills. The outcome of this assessment is an overview of which platforms cover the feature, training, and inference pipelines in full, as well as operations, today, and it determines which platforms are carried forward into the experiments.

The comparison phase answers RQ2 by running the benchmark against two of the highest scoring platforms, one of American and one of European origin, with four models spanning both origins and two scales, namely the American models Claude Opus and Claude Sonnet, and the European models Mistral Large and Mistral Medium. For each model and platform, a treatment grid crosses both interfaces with both skill conditions across all tasks, so that agents of different origins and scales can be compared on platforms of different origins under identical tasks and grading.

The optimization phase addresses RQ3 by using a coding agent to generate first six optimized variants of the European platform's CLI, each isolating a single design intervention, and then an optimized set of skills. The evaluation phase runs both, individually and in combination, against the same benchmark to compare them with the unoptimized CLI and the native

SDK baselines, completing the answer to RQ3.

This chapter begins by introducing experiments in software engineering (SWE) as a research method and defining the terminology and process used throughout this study. It then describes the benchmark generation phase, whose tasks form the objects of study, the market assessment phase, which answers RQ1 and determines which platforms can host the evaluation, and the scoping of the experiments that realize the comparison, optimization, and evaluation phases. The planning phase that follows describes the meta-harness, the interfaces and skills under comparison, the independent and dependent variables, the hypotheses, the experiment design as a set of committed treatments, the execution protocol, the stochasticity controls, the instrumentation, the analysis procedure, and the four dimensions of validity. The chapter concludes with the operation of the experiments, covering the implementation and preparation of the meta-harness, the execution of runs, and the validation of the collected data. The analysis and interpretation of the results, which applies descriptive statistics and formal hypothesis tests to the data and interprets the findings in light of the three RQs, follows in the next chapter.

3.1 Experiments in Software Engineering

An experiment in software engineering (SWE) is defined as "an empirical enquiry that manipulates one factor or variable of the studied setting. Based on randomization, different treatments are applied to or by different subjects, while keeping other variables constant, and measuring the effects on the outcome variables. In human-oriented experiments, humans apply different treatments to objects, while in technology-oriented experiments, different technical treatments are applied to objects" [222]. Experiments are used to evaluate beliefs such as a theory or hypothesis by testing the relationship between a treatment and an outcome, thereby allowing researchers to draw conclusions about cause and effect [223].

Central to any experiment is the distinction between two types of variables [222]. Independent variables, also known as factors, are the variables in the process that are controlled and manipulated by the experimenter. Dependent variables, also known as response variables, are those whose values are observed and measured to study the effect of changes in the independent variables. An experiment studies the effect of changing one or more independent variables while all other independent variables are held at a fixed level. A treatment is one particular value of a factor, and the choice of

which treatments to apply constitutes the experiment design. Treatments are applied to combinations of objects, the entities being studied, and subjects, the participants or tools applying or receiving the treatments, if any. An experiment consists of a set of tests, where each test is a unique combination of a treatment, a subject, and an object. The number of tests carried out directly affects the experimental error, which is the estimate of how precisely one can measure the mean effect of any experimental factor. The experimental error, in turn, determines the confidence that can be placed in the results.

The experiment process consists of five main phases [222]. The first phase is scoping, in which the goals and objectives of the experiment are established. The goal is formulated from the research problem, and a commonly suggested framework defines the scope in terms of the object of study, the purpose, the quality focus, the perspective from which the experiment is conducted, and the context in which it takes place.

The second phase is planning, in which the context is set, including the personnel involved and the environment. During planning, hypotheses are stated formally as the null hypothesis, which represents the default assumption of no effect, and the alternative hypothesis, which represents the expected effect. The independent and dependent variables are identified, and the experiment design is specified, including the instrumentation to be used. A critical aspect of planning is addressing validity, which has four dimensions. Internal validity concerns the reliability of the results and whether the observed effects can genuinely be attributed to the treatment rather than to confounding factors. External validity addresses how generalizable the results are beyond the specific experimental setting. Construct validity examines whether the treatment truly reflects the cause construct and whether the outcome adequately captures the effect construct. Conclusion validity concerns whether the observed relationship between treatment and outcome is statistically justified.

The third phase is operation, which encompasses the three activities of preparation, execution, and data validation. Preparation involves readying the subjects and materials for the experiment. During the execution phase the experiment is conducted. Data validation involves checking the collected data for correctness and completeness before analysis.

The fourth phase is analysis and interpretation. In this phase, descriptive statistics are used to summarize the data, and the researcher considers whether the data set should be reduced, for example by removing outliers. A formal hypothesis test is then performed, and the results are interpreted. If the evidence is sufficient, the null hypothesis may be rejected in favor of the alternative hypothesis.

The fifth and final phase is presentation and packaging, in which the results are documented and compiled into a complete package that allows others to understand, evaluate, and replicate the experiment.

The remainder of this chapter applies the first three phases of this process to the experiments conducted in this study. The fourth phase, analysis and interpretation, is carried out in the next chapter. The fifth phase, presentation and packaging, is realized through this thesis itself and through the public MLPlatformAgentBench (MLPAB) GitHub repository [224], which packages the meta-harness, treatments, evaluation suites, and results. Applying this process begins with the first phase of the method, the generation of the MLPAB benchmark.

3.2 Benchmark Construction

This section describes how the MLPlatformAgentBench (MLPAB) benchmark was constructed, covering the requirements that shaped its design, the generation of its tasks with a coding agent, the resulting tasks and their grading protocol, the two capstone projects derived from real world use cases, and the verification of the benchmark itself before any experiment was run. It begins with the requirements.

3.2.1 Design Requirements

Six requirements were fixed before any task was written, and every task in the benchmark satisfies all of them.

The first requirement is *ground truth by construction*. The answer key of every task is computed by committed deterministic pandas and numpy code. No large language model (LLM) produces the ground truth, and the platform under test is never used to compute it, so the correctness of the answer key does not depend on the systems being evaluated.

The second requirement is *assertion suite grading*. Every run is graded by assertions implemented in code, and the counts of passed, failed, and skipped assertions are recorded. Each failed assertion carries a named diagnosis, so a failure is attributed to a specific mistake rather than merely detected.

The third requirement is *validity checks at generation time*. Every generated instance must pass two checks before it is handed to an agent. The first check requires that the committed reference solution reproduces the ground truth exactly, which proves that the task is solvable as stated. The second check requires that every wrong answer differs from the ground truth,

where a wrong answer is the result a plausible shortcut would produce, such as aggregating over the wrong window or skipping a required filter. An instance in which a shortcut yields the correct answer cannot distinguish the shortcut from the solution, so it is rejected and the generator reseeds deterministically.

The fourth requirement is *platform-neutral prompts*. Tasks state their required outcome in domain language, and never in platform primitives, so the same prompt is valid on every platform and translating domain language into platform mechanisms is part of what the benchmark measures.

The fifth requirement is *fresh instances per run*. Every run receives a newly generated instance from a deterministic seed derived from the session, task category, task, and repetition identifiers, and instance specific resource names carry a suffix of six hexadecimal characters derived from the seed. The seeding exists so that no instance matches a task or solution already present in model training data, and so that no two runs share data or resource names. Every run nonetheless remains reproducible from its seed.

The sixth requirement is *separation of grading from the agent*. Grading is performed by code, never by an LLM judging the agent's output, which requires implementing a checker adapter for every supported platform. The private answer key, comprising the ground truth and the wrong answers, is materialized in a sibling directory outside the agent's file system boundary, and grading runs outside that boundary through these adapters, so the agent can never observe the answer and a plausible transcript contributes nothing to the grade. With these six requirements fixed, the next step was to generate a benchmark that satisfies them.

3.2.2 Task Generation

The benchmark was generated with the coding agent Claude Code running Claude Opus 4.8, operating under the experimenter's direction and against the six requirements above. Task construction followed the feature, training, and inference pipeline architecture defined by Dr. Jim Dowling in *Building Machine Learning Systems with a Feature Store: Batch, Real-Time, and Large language model (LLM) Systems* [36], which defines a machine learning operations (MLOps) paradigm that organizes a machine learning (ML) system into independent feature, training and inference (FTI) pipelines connected through a feature store. The feature, training, and inference task groups each cover the sub categories of the corresponding pipeline type, so that the benchmark operationalizes the pipelines of RQ1 rather than an ad hoc task selection.

The FTI task groups and the operations group together cover MLOps and

automated machine learning (AutoML), with operations covering use cases such as alerting, lineage, and scheduled jobs. The 22-task benchmark extends beyond MLOps tasks with LLM fine-tuning as part of the model training group and LLM serving as part of the model inference group.

Two capstone projects complete the benchmark. Derived from the example systems of Dowling's book's `mlfs-book` GitHub repository [225], they reproduce its air quality forecasting and credit card fraud detection use cases as fresh seeded instances, the former built from real Particulate Matter with an Aerodynamic Diameter of 2.5 µm or less (PM2.5) measurements joined with Open-Meteo weather archives and the latter from a deterministic synthetic transaction log [226], and each is graded by a held out predictive metric against a calibrated per instance bar, root mean square error (RMSE) and receiver operating characteristic (ROC) area under curve (AUC) respectively, together with state checks that the feature group, training dataset, and registered model exist on the platform, thereby measuring whether competence on individual pipelines composes into full pipeline competence. Together, the pipeline and operations tasks and the capstone projects form the benchmark presented next.

3.2.3 Task Overview

The benchmark consists of 22 tasks split across five groups. The feature, training, and inference groups map directly onto the feature, training and inference (FTI) pipelines, and together with the operations group they cover machine learning operations (MLOps), while the automated machine learning (AutoML) additions, large language model (LLM) fine-tuning and serving, sit inside the training and inference groups. Each task within each of these groups adds one realistic complication that the agent must navigate, such as overlapping redelivered export files whose rows must land exactly once, out of order corrections, or schema breaking reloads. There are no difficulty tiers. Table 3.1 describes the 22 tasks.

Table 3.1: Short description of each of the 22 MLPlatformAgentBench (MLPAB) tasks. Every task states its requirement in domain language and is graded by reading the deliverable back from the platform. Full per-task documentation, including the generator's notes, an example prompt, the staged files, and the enumerated assertion suite, is available in the GitHub repository accompanying this thesis [227].

Task	**Description**
Feature	
`ingest`	Register a feature table and load a full export whose two files overlap so that every row lands exactly once.
`backfill`	Load out of order batches containing corrections so that the table holds each row's latest revision.
`mit`	Compute a model-independent transformation, a per account rolling seven day sum with exact window semantics, into a derived feature table.
`validate`	Enforce documented data constraints, loading only clean rows and reporting every rejected row identifier.
`incremental_load`	Load all daily increments and register a recurring scheduled ingestion job on the platform.
`full_reload`	Recreate a feature table after a breaking upstream schema change instead of merging the new export into the old data.
`training_data`	Build a point in time correct training dataset across four feature tables with late arriving rows, duplicates, and one table whose rows after the label time strongly encode the label, so that ignoring the as of cutoff leaks it.
Training	
`train`	Run a provided deterministic training script as a platform job and land its predictions in an online enabled feature table, bit exact.
`mdt`	Apply a model-dependent transformation with statistics fitted on the training split only, never on the serving data.
`register`	Register a provided model artifact in the platform's model registry with the exact provided metrics attached.

continued on next page

Table 3.1: Short description of each of the 22 MLPAB tasks, continued.

Task	Description
`llm_finetuning`	Run a provided deterministic fine-tuning script as a platform job and register the resulting model with the exact metrics the script produced.
Inference	
`batch`	Batch score a whole entity population with point in time correct feature values through an as of join that must not look past the cutoff.
`online`	Materialize features for online serving and retrieve them through the platform's low latency read path rather than from the source data.
`odt`	Compute an on-demand feature at request time from request parameters joined with a stored profile, applying the exact formula.
`llm_serving`	Deploy a provided model as a real time endpoint and invoke it correctly over the platform's inference path.
`recsys`	Run a two-tower retrieval and ranking step with dot product relevance, exclusion of already seen items, and a deterministic tie break.
`vector_search`	Load embeddings into the platform's vector capable store and run its native similarity search with the correct Euclidean metric.
Ops	
`scheduled_jobs`	Set up a recurring scheduled job on the platform from a provided script and prove that it ran.
`alerting`	Create a job that is guaranteed to fail, attach an alert to that failure, and report the configuration.
`lineage`	Build a derived feature table from two partially overlapping sources, register the derivation, and answer a lineage question.
Capstone	
`ccfraud`	Build the full feature, training, and inference pipeline for credit card fraud detection and publish per transaction fraud probabilities that beat a calibrated receiver operating characteristic (ROC) area under curve (AUC) bar.

continued on next page

Table 3.1: Short description of each of the 22 MLPAB tasks, continued.

Task	Description
`airquality`	Build the full feature, training, and inference pipeline for Particulate Matter with an Aerodynamic Diameter of 2.5 µm or less (PM2.5) forecasting and publish per day predictions that beat a calibrated root mean square error (RMSE) bar.

How each task is graded depends on the kind of deliverable it declares, which is defined next.

3.2.4 Grading Deliverables

Each task declares one of three deliverable kinds, which determines how it is graded. A *table* deliverable is a feature table on the platform, read back through the checker adapter and compared against the ground truth on schema, row count, and content. A *dataset* deliverable is a versioned training dataset read back the same way. A *platform* deliverable additionally requires platform state, such as a registered model, a recurring job, a real time endpoint, or an alert, verified through the adapter's state reads. One checker adapter per platform implements the assertions through which grading occurs, retrieving feature tables, reading rows and training datasets, and reading platform state such as registered models, jobs, deployments, and alerts. Grading therefore inspects the state the agent produced on the platform, through the platform's own data paths, rather than the agent's transcript or local files. The grading report records a per assertion result, and when a delivered value matches the precomputed result of a plausible shortcut, the report names that shortcut as the failure diagnosis.

Together, the deliverable kinds and grading through the platform's own read paths guarantee that the platform was actually invoked. A deliverable can only pass its assertion suite if it exists on the platform, so a locally computed result, a fabricated artifact, or a plausible looking transcript earns nothing. This closes one of the two paths by which an agent could bypass the platform, and the second path, producing the deliverable locally in the first place, is closed at execution time by enforcement. Reconciling this restriction with agent autonomy was among the hardest problems of the benchmark's construction. The agent must be prevented from working around the platform, since a task rescued with local computation would measure the agent rather

than the interface, while at the same time retaining the highest possible degree of autonomy in planning its approach, discovering the interface's capabilities, recovering from errors, and deciding how to solve the task. Instructions alone proved insufficient during development, since an agent facing a failing interface readily rescues the task with local computation, so the constraint was moved out of the prompt and into the environment, where grading by state verifies the outcome, the allowlist enforces the path the agent is on, and everything within that boundary is left entirely to the agent. Before any agent was graded, the benchmark itself was verified.

3.2.5 Benchmark Verification

The benchmark was verified in three ways before any committed experiment was run. First, the validity checks operate on every generated instance, so a defective instance can never reach an agent. Second, the generators' self tests and the meta-harness's unit test suite exercise the generators, graders, and checker adapters, so grading is tested independently of any agent.

Third, because the benchmark was generated with Claude Opus 4.8 while the experiments compare Anthropic and Mistral models, the benchmark, including the task generators, graders, and prompts, was further checked for errors with an agent from the other vendor. Mistral Vibe executing Mistral Large 3 was run over the meta-harness and the task implementations as an independent review pass, and the defects it surfaced were fixed before any committed treatment was executed.

With the benchmark constructed, the next section assesses which platforms cover the machine learning operations (MLOps) and automated machine learning (AutoML) tasks it operationalizes and can therefore host the evaluation.

3.3 Market Assessment

Before any experiment can compare interfaces, RQ1 requires identifying a broad candidate set of machine learning (ML) platforms, which interfaces they offer, and to what extent those interfaces cover the feature, training, and inference pipelines as well as operations. The market assessment answers this through a criteria-based scoring of 39 candidate ML platforms.

The set of candidate platforms is drawn from three sources. The first source is the vendor landscape reviewed in the related work chapter [181]. The second source is the platform directory maintained at featurestore.org [228],

a catalog of feature store and ML platform vendors maintained by one of the assessed vendors. The third source is a monitoring of platform announcements on Hacker News and LinkedIn between 30 January 2026 and 15 July 2026, which captures platforms and interface releases that postdate the literature.

Each platform is assessed against 31 criteria. Two criteria capture availability, namely whether the platform is actively maintained and whether it is offered as a public software as a service (SaaS). Four criteria record whether each of the three agent-facing interfaces and ML platform agents exist at all, and the documenting source is recorded as evidence. For this assessment, an *ML platform agent* is a vendor-specific agent supplied with the product, through which the platform is operated natively. A *meta-harness* developed by the platform vendor running inside or outside of the platform is scored separately, as is *agent deployments*, which records whether the platform documents hosting for agents. An agent the vendor ships that meets neither definition is recorded as an *other software agent* without score. The core of the assessment maps the MLPlatformAgentBench (MLPAB) benchmark directly onto the platform. For each of the four, the Python software development kit (SDK), the command line interface (CLI), the Model Context Protocol (MCP) server, and ML platform agents, the assessment scores whether it covers each of the four benchmark stages, namely the feature, training, and inference pipelines and operations, yielding sixteen stage-coverage criteria derived from those tasks. The remaining criteria are hosting support for MCP servers, recorded as a standalone criterion, the ML platform agent capabilities of running PySpark code, accessing a file system, and providing dashboards, a documented meta-harness, documented agent deployments, and the availability of agent skills. In addition, the headquarters location of each vendor is recorded so that platforms of European and American origin can be compared under RQ2.

All criteria are documented from public documentation and source code repositories. The coverage of the SDK, CLI, and MCP interfaces was assessed using the coding agent Claude Code running Claude Opus 4.8, which was directed at each platform's documentation and source repositories to probe the interface surface, with every judgement reviewed by the experimenter and the supporting source recorded alongside it. ML platform agents were assessed from documented vendor-agent capabilities or documented meta-harnesses under the same review rule. The ML platform agent criteria are credited only if a documented source actively mentions a vendor-built agent performing the corresponding platform work, and the meta-harness criterion only if the vendor documents a harness that hosts general-purpose

coding agents in the platform. An external agent that merely reaches a generic public application programming interface (API) without any native platform support does not satisfy the definition.

Each criterion is scored as fully supported, partially supported, or unsupported, based on the documented capabilities. From these 31 recorded criteria, the market coverage score reported in the results chapter aggregates 23 coverage criteria. These are the four stages each of the SDK and the CLI, the four stages plus documented server hosting of the MCP, the four stages plus the PySpark, file system, and dashboard capabilities of ML platform agents, and one criterion each for a documented meta-harness, documented agent deployments, and a published set of skills. Full support contributes one point and partial support half a point. The four criteria recording whether each interface exists, the availability and headquarters facts, and an other software agent are not scored. A platform therefore scores high exactly to the extent that its interfaces and agents can carry out the benchmark.

The scores are aggregated into a coverage score per platform, computed as the number of fully supported scored criteria plus half the number of partially supported ones, divided by the 23 scored coverage criteria. The leading platforms are those that are actively maintained, offered as a public SaaS, and score at least 35%, a rule applied uniformly to every assessed platform. The resulting coverage matrix constitutes the answer to RQ1 and is reported in the results chapter. It also determines the platforms carried into the experiments. The benchmark is run against the highest scoring platform overall and the highest scoring European platform, both of which provide a CLI and a Python SDK with full coverage of the feature, training, and inference pipelines as well as operations, and which therefore support the origin comparison of RQ2 without confounding platform origin with interface coverage.

The assessment has methodological limitations that bound what the answer to RQ1 can claim. The scoring is a single pass by one coding agent with author review, so no inter-agent reliability can be reported, and the judgements inherit the accuracy of public documentation, which can lag or overstate actual capability. The equal weighting of the 23 scored coverage criteria and the half credit for partial support are modeling choices without an empirical basis, so the tier boundaries should be read as a coarse ordering rather than a precise ranking. Finally, using Claude Opus 4.8 to assess interfaces that Claude models later operate introduces a mild circularity, which is limited by the fact that the assessment scores whether documented coverage exists rather than how well an agent performs with it. The candidate set is purposive rather than exhaustive. Its source lists and news monitoring provide breadth, but

there is no reproducible population frame, formal inclusion rule, or stopping criterion, so counts apply only to the assessed set.

With the market assessed, the next section scopes the experiments that answer the second and third research questions (RQs).

3.4 Scoping the Experiments

The overall goal of the experiments is to empirically compare command line interface (CLI)-based and software development kit (SDK)-based interfaces for coding agents operating machine learning (ML) platforms, to study how the use of skills affects performance under each interface type across agents and platforms of different origins and scales, and to evaluate whether a coding agent can improve its own operating interface by optimizing the platform CLI and its accompanying skills. The experiments require no human participants to apply treatments. They are technology-oriented.

The objects of study are the 22 tasks of the MLPlatformAgentBench (MLPAB) benchmark, spanning the feature, training, and inference pipelines, extended by large language model (LLM) serving and fine-tuning and complemented by operations and capstone projects, instantiated as fresh, seeded task instances on one European and one American platform selected through the market assessment.

A subject of study is the combination of a coding agent and an underlying LLM. The meta-harness supports three coding agents, Claude Code executing Anthropic models, Mistral Vibe executing Mistral models, and Open Artificial Intelligence (OpenAI) Codex executing OpenAI models. Codex is supported but is not part of the committed experiments, since the origin comparison contrasts European and American agents, and OpenAI Codex, like Claude Code, is an American stack, so including it would have added cost without adding an origin contrast.

Each vendor's models are accessible only through that vendor's agent, so model and coding agent cannot be varied independently, and a subject is in practice the deployable combination of model and coding agent. This coupling is deliberate, since the stack is also the unit a practitioner usually chooses between, but it means that model contrasts are stack contrasts, a consequence discussed under validity.

The experiments evaluate four models chosen to span origin and scale, each pinned to an explicit version-locked identifier such as `claude-opus-4-8`. The American models are Claude Opus 4.8 as Anthropic's flagship model and Claude Sonnet 4.6 at the medium

scale, and the European models are Mistral Large 3 as Mistral's flagship model and Mistral Medium 3.5 at the medium scale. These four enable the origin and scale comparison-demanded by RQ2.

The treatments correspond to the manipulated factors, namely the interface through which the agent operates the platform, either the CLI or Python SDK, the set of skills provided to the agent, none, skills, or optimized, and the optimization state of the CLI, the unoptimized CLI or one of six agent-optimized variants. The purpose is comparative, measuring differences in accuracy, cost, and turns between interface types, skill conditions, models, and platforms, and evaluating the effect of agent-driven interface optimization against the unoptimized CLI and native SDK baselines.

With the goal, objects, subjects, treatments, and purpose established, the next phase specifies how the experiments are structured to deliver on this scope.

3.5 Planning the Experiments

The planning phase defines how the experiments are structured to answer the second and third research questions (RQs) in a controlled and repeatable manner. It specifies the experimental setup, benchmark, variables, hypotheses, procedures, and instrumentation, ensuring that observed differences can be attributed to the manipulated factors rather than to external variation.

Planning begins by describing the meta-harness in which the experiments are conducted, including the infrastructure on which it runs and the components that make up the environment under evaluation.

3.5.1 Meta-Harness

The experiments are conducted within MLPlatformAgentBench (MLPAB), as shown in Figure 3.1, a meta-harness for evaluating coding agents operating remote machine learning (ML) platforms. It is primarily organized into four component groups, `configs`, `evals`, `treatments`, and the `mlpab` package.

The meta-harness defines the environment under evaluation, the subjects, and the objects of study, and provides the infrastructure required to execute repeated runs under consistent conditions.

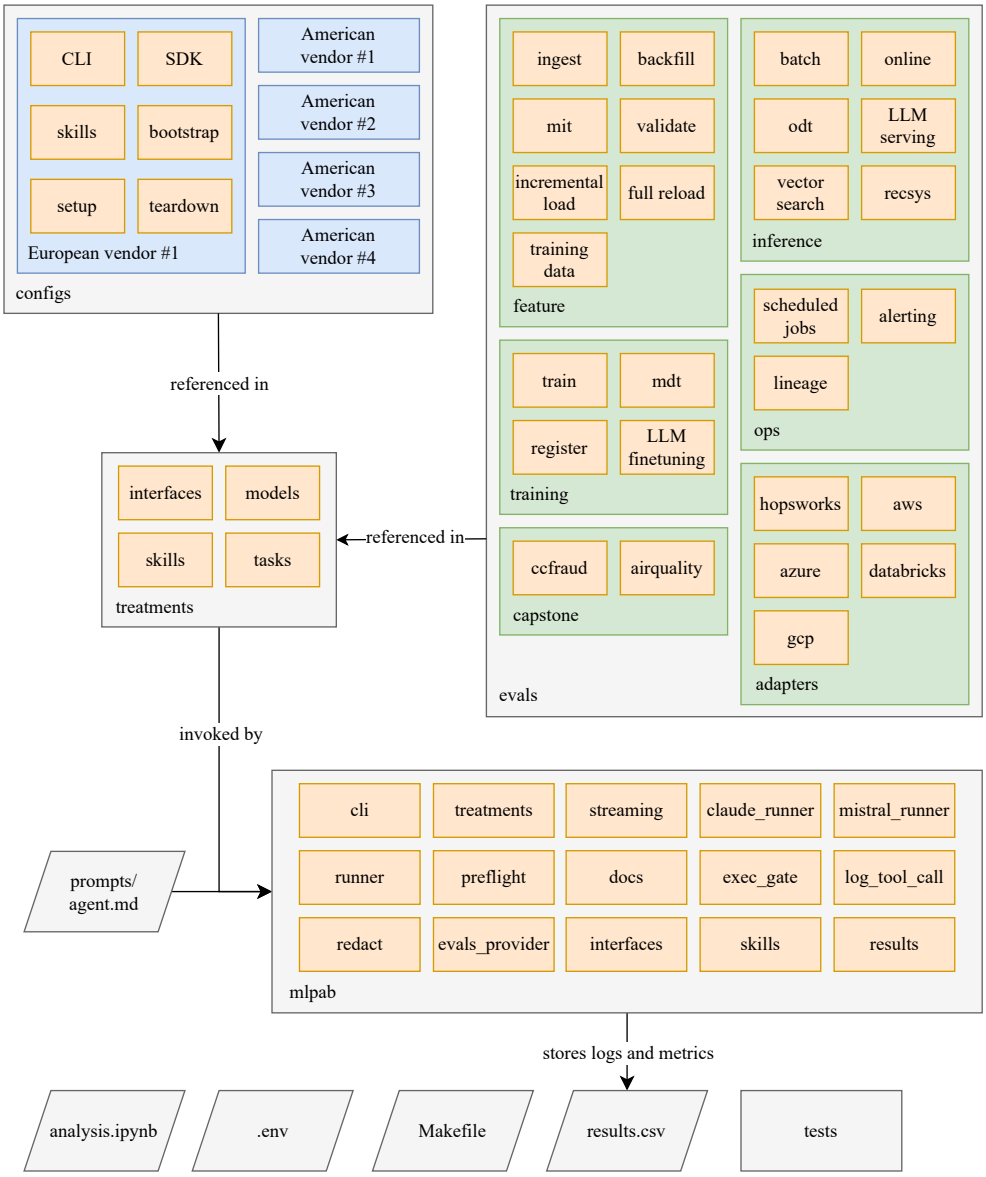

Figure 3.1: Architecture of the MLPAB meta-harness

The `configs` group holds one *platform definition* per supported platform. Each definition comprises one manifest per interface declaring where the interface source is hosted, the pinned commit at which it is built, how it is installed, authenticated, and tested, the credential keys it requires, a version string written into the results, and the short prompt fragment that introduces the interface to the agent. Alongside the interface manifests, each definition contains a `skills` manifest pointing at the published set of skills at a pinned commit, a `bootstrap` step that prepares the platform account, and

`setup` and `teardown` scripts that provision and remove platform resources around each run, each supporting a verification mode.

The `evals` group defines the objects of study. It contains feature, training, inference, and operations tasks together with capstone projects, represented by `feature`, `training`, `inference`, `ops`, and `capstone`. Each task is implemented as a generator and grader pair, together with one *platform adapter* per platform implementing the platform reads through which grading occurs. Platform definitions and evaluation suites are referenced in *treatments*, numbered flat YAML Ain't Markup Language (YAML) files that each bind a model, an authentication mode, a repetition count, a per-run time budget, a task list per category, a list of interface manifests, and a list of skill conditions, and expand deterministically into the Cartesian grid of runs over tasks, interfaces, and skill conditions. Each committed treatment is reported in the results chapter, and its outputs are stored under a matching directory in the results store, keyed by model, platform, interface, interface version, skill condition, task, and repetition index.

Treatments are invoked by the `mlpab` Python package, which coordinates every run. Its `cli` module exposes the commands through which treatments are started, monitored, and resumed, and its `treatments` and `runner` modules expand a configuration into its run grid and move each run through its lifecycle. Before any work begins, the `preflight` module verifies everything the treatment needs and aborts if anything is missing. It builds any missing interface artifact from its pinned source, verifies authentication against the live platform, and statically verifies every skill bundle; for Claude treatments it additionally invokes a short probe session to confirm that the skills are visible to the agent. The `interfaces` and `skills` modules then install the interface under test and inject the skills where the condition requires it, and the `docs` module strips vendored agent instructions, such as a source repository's own `CLAUDE.md` file or `.claude` directory, from every materialized interface source, so that no directive from outside the treatment reaches the agent. One runner per coding agent, `claude_runner`, `mistral_runner`, and `codex_runner`, wraps each agent in a sandboxed, policy-enforced session behind the same application programming interface (API), which is what makes MLPAB a meta-harness in the sense adopted in the background, while a shared agent prompt template, `prompts/agent.md`, is assembled into every run so that task framing is identical across agents, models, and conditions, differing only in the interface fragment, in conditional blocks for the platform-under-test and local-baseline modes, in the detected host environment, and in the time budget. During a run,

the `exec_gate` and `log_tool_call` modules enforce the sandbox. The latter runs as a pre-tool-use hook for the agent that supports one, deciding each call before it executes and appending one record per call, while the former applies the same enforcement logic as an intercepting shell for agents that expose no hook mechanism, so that the confinement holds for every agent. The `streaming` module captures the agent's output as it is produced. Once the agent stops, the `evals_provider` module, which generates the seeded task instance before the agent starts, executes the grading against the platform state, the `redact` module removes credentials from all persisted artifacts, and the `results` module appends one row per run to the results store. Supporting artifacts complete the meta-harness. A `.env` file supplies credentials, a `Makefile` automates installation and verification, and a unit test suite under `tests` guards the `mlpab` package itself.

Finally, the `results` store collects the logs and metrics of every run. One row per run is appended to the single results table, a comma-separated values (CSV) file at `results/results.csv`, and per run artifacts, including the agent log, the per command record, the grading report, and any local submission, are stored under the treatment's results directory. Jupyter notebooks stored alongside the results perform the subsequent analysis.

With the meta-harness in place, the next planning element is the pair of interfaces under comparison and the skills that accompany them.

3.5.2 Interfaces and Skills

For each platform, the two interfaces under comparison is a versioned interface based on the platform's command line interface (CLI) and Python software development kit (SDK). The native SDK is the natural baseline demanded by RQ3. One capability parity correction was required because the European platform's CLI lacked the alerting command group present in the SDK, which would have confounded interaction style with interface coverage on the alerting task, so the missing command group was implemented and contributed upstream. The pinned CLI under test is version `5.1+alerting` from the contribution fork, while the SDK is version `5.0.0` from a pinned upstream commit. The same capability review found no task coverage gap between the pinned interfaces of the American platform. The introduction of each interface to the agent is deliberately minimal, naming the command or module and its authentication and directing the agent to discover capabilities on-demand, for the CLI through `-help` and for the SDK through Python introspection.

The skills factor has three levels. In the *none* condition, no skills are installed. In the *skills* condition, the meta-harness installs the skills published by the platform vendor, fetched at a pinned commit, in the run's project skills directory. In the *optimized* condition, introduced in the evaluation phase and evaluated with Claude Code, the agent receives a set of skills authored with a coding agent from the comparison phase run artifacts.

The RQ3 further asks whether a coding agent can improve the CLI itself. To this end, six optimized CLI variants are generated with Claude Code running Claude Opus 4.8 from the comparison phase run artifacts, each branching off the pinned baseline CLI and adding exactly one design intervention. Each variant is built into its own build home from its own pinned commit, carries a distinct version string in the results, and is accounted and graded as a CLI run, so variants are compared against the baseline through the version column under otherwise identical conditions. For the combined optimizations, each variant is paired with a copy of the optimized set of skills extended by one skill documenting the variant's added capability.

With the interfaces, skills, and optimization artifacts in place, the next step is to define the variables that the experiments manipulate and measure.

3.5.3 Variables

The independent variables are *interface type*, with the two levels command line interface (CLI) and Python software development kit (SDK), *skill usage* with the three levels none, skills, and optimized, of which the optimized level applies only to the evaluation phase, *CLI optimization* with the unoptimized CLI and six optimized variants as levels, *model*, with four levels spanning the American models Claude Opus and Claude Sonnet and the European models Mistral Large and Mistral Medium, and *platform*, with two levels spanning the European platform and the American platform. The *task* is varied across the 22 tasks of the benchmark, and each task contributes one cell per condition.

The dependent variables are *accuracy*, measured per run as the binary success indicator and the fraction of assertions passed by the read back deliverable, *cost*, measured as the token consumption of the run's model invocations and as their monetary cost in United States Dollar (USD), both derived from the coding agent's token accounting, and *turns*, measured as the number of model invocations within the run, decomposed further into per category tool call counts, most importantly the number of interface calls. This cost is the model invocation charge only. Platform compute, storage, warehouse, cluster, and networking charges are outside the mea-

surement. Because USD prices reflect vendor pricing decisions rather than computational effort, cross-vendor cost comparisons carry this caveat, and USD is reported as the practically relevant model charge, with token counts recorded for descriptive comparison. Since each run's assertion fraction is already normalized within its task, averaging the fraction across tasks weights every task equally regardless of how many assertions it contains. *Latency* is recorded as an auxiliary measure and decomposed so that the local time records the time spent using the interface on the client side, everything beyond that within an interface call is attributed to the platform and recorded as platform time, and time spent waiting on provider rate limits is recorded separately. Platform-side latencies are subtracted in `results.csv` so that they do not contaminate the interface latency measures, since they reflect platform load rather than interface behavior, and the same holds for rate limit waiting, which reflects provider load. A *validity* indicator records whether the run produced a gradable outcome at all.

All other factors, including the task prompt template, coding agent configuration, sandbox contents and enforcement policy, platform provisioning, the per-run time budget, and grading logic, are held constant across conditions to isolate the effects of the independent variables, with one documented exception, the command sandbox exclusion of the American platform's CLI binary described in the execution protocol. These variables enable the formal statement of the hypotheses that the experiments test.

3.5.4 Hypotheses

Two hypotheses follow from the research questions. The first hypothesis addresses RQ2 and concerns the interface and skills comparison, and the second hypothesis addresses RQ3 and concerns the interface optimization evaluation. The RQ1 is answered by the market assessment rather than by a hypothesis test.

The first hypothesis, for the interface and skills comparison, states as its null form that there is no difference in accuracy, cost, turns, or time between the command line interface (CLI) and software development kit (SDK) interfaces, between the with skills and without skills conditions, between models of different origins and tiers, or between platforms of different origins. Its alternative form states that at least one of these factors produces a systematic difference in at least one of these metrics. These composite hypotheses are not tested as a single omnibus test but are evaluated through planned contrast groups, each with its own paired test. For the interface contrast the expectation is directional. The progressive

disclosure of the CLI is expected to reduce token cost and turns relative to the SDK, where the agent writes code from its prior knowledge of the module and discovers mismatches through runtime errors, and to improve speed as reflected in the auxiliary latency decomposition.

The second hypothesis, for the interface optimization evaluation, states as its null form that the agent-optimized CLI variants and the optimized set of skills do not change agent performance relative to the unoptimized CLI with no skills or with the set of skills, and do not close the gap to the native SDK baseline. Its alternative form states that optimization changes performance, and that the effect may vary across optimization variants, their combination with optimized skills, and the feature, training, inference, operations, and capstone projects.

Testing these hypotheses requires a design that arranges the variables into a concrete set of runs.

3.5.5 Experiment Design

The experiments realize the last three phases of the method in sequence. The comparison phase executes treatment grids that cross both interfaces with the none and skills conditions across all tasks, for multiple models and platforms, and produces the data for RQ2. The optimization phase produces first the six optimized command line interface (CLI) variants and then the optimized set of skills with Claude Code running Claude Opus from comparison phase run artifacts. The evaluation phase executes the CLI optimization variants and the optimized set of skills on the European platform and produces the data for RQ3, evaluating first the optimized CLI alone and then its combination with the optimized skills. The optimization condition is therefore a between phase factor rather than a within phase factor.

The design is realized as 21 numbered treatments that each expand into a run grid, eight in the comparison phase and thirteen in the evaluation phase. A cell of a grid is one combination of platform, model, interface, skill condition, and task, and every committed cell is planned to run exactly once. Each run executes a fresh seeded task instance with its own provisioning. Statistical power derives from pairing conditions across the 22 tasks rather than from repeated runs of the same cell. This is a deliberate trade-off that spends the available budget on breadth across tasks, models, and platforms rather than on repetition within cells. Each run is bounded by a one hour compute budget, measured as elapsed time excluding rate limit waiting. Runs within a treatment execute sequentially in the order of the deterministic

expansion. Treatments themselves can execute in parallel because every run is isolated in its own platform namespace. Completed cells are never re-executed and are skipped when a treatment is restarted. Cells without a completed attempt are re-run only on an explicit retry, which purges the failed attempt and executes the cell cleanly. The schedule therefore remains reproducible and completed results are never overwritten. The three phases are described in turn, beginning with the comparison.

3.5.5.1 Comparison

The comparison phase comprises the grid treatments on the two selected platforms. Treatments 1 to 4 cross both interfaces with both skill conditions across the task templates, for Claude Opus, Mistral Large, Claude Sonnet, and Mistral Medium. Treatments 18 to 21 execute the identical grid for the same four models on the other platform. The planned comparisons are evaluated on the 22 benchmark tasks: interface and delivered-skills effects within each model and platform, model-stack differences within each platform, and differences between the earlier European-platform runs and the later American-platform runs within each model. This phase produces the data used to test the first hypothesis, and its run artifacts, in particular the agent logs and per command records of the European platform's CLI runs, result in the construction of the optimization artifacts in the optimization phase.

3.5.5.2 Optimization

In the optimization phase, first the six CLI variants and then the optimized set of skills are produced for the European platform. Comparison phase CLI runs on the European platform are analyzed through their per-run artifacts to identify recurring sources of failure and of token, turn, and wall clock inefficiency, such as one login and one process start per invocation, verbose or unstable output that the agent must re-parse, and retries that duplicate state. Each identified pattern is addressed either by one CLI intervention, kept isolated on its own branch so that its effect is attributable, or by the optimized set of skills, which is kept general and free of task-specific content so that it cannot encode solutions to the benchmark. The optimization itself is performed by Claude Code running Claude Opus 4.8, operating on the interface source and the skills text.

The direction given to the agent is documented by the prompts below, quoted verbatim except that vendor names have been redacted, since the platform identities are reported only in the next chapter, and that one

task name was removed as it did not make the final benchmark after all. An exploratory prompt of 19 June 2026 considered both interfaces before the optimization was narrowed to the CLI.

> *Consider the results for [the] CLI and software development kit (SDK). How would you optimize both interfaces to improve task success, reduce token usage and time spent on tasks.*

The CLI optimization was directed on 22 June 2026 through a proposal prompt and, after the agent had proposed the six interventions, an implementation prompt.

> *Given the [...] CLI, and the results, what optimizations do you propose to reduce cost_usd and wall_time_s without overfitting the CLI to the benchmark.*

> *implement all six changes in individual commits, generate six additional cli.yaml files with the references and clear version names — then generate six new treatments with opus and a slim selection of tasks, i.e. ingest, train, batch, [...], ccfraud and airquality*

The skills optimization was directed on 23 June 2026 through a single prompt.

> *Do a second run using the baseline and optimizations. This time, add skills. Use the original [...] skills for the baseline, add a second baseline using optimized skills for the baseline, and optimize the skills for each code optimization as well. The goal remains to reduce local time, reduce cost, and improve the assert rate.*

One scoping consequence must be stated plainly. The optimization inputs are runs on the same task templates that the evaluation phase runs on, and fresh seeds change the data and resource names of an instance but not the structure of its template. Whether the artifacts deliver this improvement is measured in the evaluation phase.

3.5.5.3 Evaluation

The evaluation phase evaluates the optimization artifacts on the European platform with Claude Opus, holding the model, platform, tasks, and grading identical to the comparison phase, and follows the two optimization steps in sequence, first the optimized CLI alone, then the optimized CLI together with the optimized skills. Treatments 5 to 10 evaluate each of the six CLI variants without skills, isolating the effect of each intervention against the unoptimized no skills CLI cells of treatment 1. Treatment 11 then evaluates the unoptimized CLI with the optimized set of skills, isolating the skills optimization effect against both the no skills and the skills CLI cells of treatment 1. Treatments 12 to 17 finally combine each CLI variant with each of its extended optimized set of skills, testing whether the two kinds of optimization compose. Across all evaluation phase comparisons, the native SDK cells of treatment 1 serve as the additional baseline.

With the design in place, the next planning element is the protocol that governs each individual run.

3.5.6 Execution Protocol

The entire protocol is automated by the meta-harness. It starts a treatment with a single command, `mlpab start <treatment>`, where the treatment references its numbered YAML Ain't Markup Language (YAML) configuration file, which in turn references the interface configurations, and no manual step occurs between the start of a treatment and the persistence of its results.

Each run proceeds through a fixed sequence of steps enforced by the meta-harness. First, a fresh task instance is generated from the run's deterministic seed. The agent-visible inputs are staged into the run directory, while the answer key is placed outside the agent's boundary. Second, the platform is provisioned for the run in its own namespace, a dedicated project or a per-run dataset and resource prefix, depending on the platform's isolation primitive, and a verification step confirms that the platform is ready. The interface's authentication is verified per-run. Third, the per interface virtual environment is cloned read-only into the run's sandbox, the configured set of skills is placed in the run environment for a skill condition, and the coding agent is started inside the sandbox with the assembled prompt.

The prompt instructs the agent to complete the task in a single attempt. The agent is a probe of the interface, may recover from errors, but is told not to iterate over alternative approaches to polish a finished deliverable, and, if the interface lacks a required capability, to state the missing capability

plainly and stop rather than build a workaround. This single attempt protocol reflects the measurement goal of the benchmark, since a partial or low quality result produced through the interface is valid experimental data about that interface. A run in which the agent correctly stops and reports a missing capability is handled explicitly. Since no gradable deliverable exists, it is recorded as invalid and scored as zero assertions in the accuracy aggregation like any other invalid run, but unlike a crashed or timed out run, whose counts are zeroed because partial figures would mislead, its token, turn, and denial metrics are preserved and the reported gap is kept in the run's error field and artifacts, so interface coverage gaps can be separated from agent failures during analysis rather than being conflated in the accuracy metric. The constraint is enforced, not merely requested.

An allowlist rejects any command outside the interface under test and a small set of basic shell utilities for inspecting the task inputs and writing the local submission files, while local machine learning (ML) libraries, other interpreters, and, in the command line interface (CLI) condition, all local Python are rejected before execution with an explanatory denial message.* For Claude Code the allowlist is applied through the coding agent's tool use hook. For the hook-less coding agents the same policy is interposed as the shell through which they execute commands, so all coding agents face an identical boundary, and every denial is recorded. This turns the platform usage constraint from an instruction the agent might ignore into a property of the environment. For the Claude runs, one further layer is active. Claude Code ships a built-in command sandbox, and the meta-harness enables it with its network gating configured per platform and its file system gating left open, since file protections are already enforced through the agent's permission rules, which deny access paths such as the answer key, and through the hook. Transient provider errors such as server-side connection closures and model rate limits are retried with back-off, the time spent waiting is recorded separately, and the waiting extends the deadline rather than drawing on the budget, so provider load can neither truncate a run nor contaminate the latency measures.

Fourth, after the agent terminates, the checker adapter reads the deliverable back through the platform's own data paths and executes the task's assertion suite, producing the grading report. This protocol describes all 22 tasks of the benchmark. Deliverable reads are attempted exactly once and are not retried. A live reliability probe of the platform's query service during preparation established that a read failure on a healthy service means the deliverable was

*The full allowlist is reproduced in Appendix A.

never produced, so a failed read is recorded with its underlying reason and graded as a missing deliverable rather than retried, and the preserved reason allows any residual platform-side read anomaly to be identified during data validation. Fifth, teardown removes the run's platform namespace, and a verification step confirms that nothing was left behind, so that subsequent runs start from a clean platform. Finally, the meta-harness appends one row for the run to `results/results.csv` and stores the run's artifacts.

Beyond the execution protocol, additional measures are taken to limit variation introduced by the underlying large language model (LLM) and the environment.

3.5.7 Stochasticity Controls

Because the agents are driven by large language models (LLMs) accessed through their vendors' coding agents, run-to-run variation cannot be eliminated but is constrained as far as practical. Model versions are pinned through explicit version-locked identifiers, interface builds and sets of skills are pinned to fixed commits and content hashes, and task generation is seeded, making those inputs to a run reproducible. The coding agent executables themselves were not pinned. Interfaces are built locally from source at their pinned commit rather than installed from a package index, so the interface under test never drifts between runs, and where a provider does not publish pricing for a pinned model, the price per token is fixed in the treatment so that reported costs remain authoritative. The prepared virtual environments are built once and cloned read-only, eliminating dependency drift within and across runs. The prompt template, sandbox layout, enforcement policy, and grading logic are identical across the runs of a condition, and the preflight probes ensure that runs are only started against built interfaces, reachable platforms, and responsive models. Vendored agentic coding manifests shipped inside interface source repositories, such as project memory files, are stripped at build time so that no vendor-authored directives leak into the agent's context outside the controlled skills factor.

Residual non-determinism nevertheless remains, arising from sampling in the hosted models, whose decoding parameters are controlled by the coding agents rather than the experimenter, from batching effects on shared inference infrastructure, and from platform-side latencies that alter the observation sequence the model conditions on. Under the single run-per-cell design, this residual variation is not estimated within cells. Instead, conclusions are drawn from effects that are consistent across the 22

benchmark tasks, which reduces reliance on any individual observation but does not estimate or bound run-to-run variation.

The data needed to evaluate each run is captured by the instrumentation built into the meta-harness.

3.5.8 Instrumentation

All runs are logged automatically by the `mlpab` package, the coding agents, and the checker adapters, with the package acting as the central aggregation point. Token and cost figures are derived from the per-run session transcripts retained in the artifacts, summing the usage of every model invocation including subagent calls and pricing prompt cache writes and reads at the published multipliers, so the recorded cost is the cache-inclusive cost. Per token prices are pinned in the treatment where the provider does not publish them. Tool invocations are recorded one record per command, by two paths that differ in how directly the record is tied to the enforcement. For the agent that supports a pre-tool-use hook, the enforcement hook writes the record itself, so the counted interface calls are, by construction, exactly the calls the enforcement permitted. For agents that expose no hook, where the same enforcement logic runs as an intercepting shell that records the denials, the command log is reconstructed from the run transcript after the run, so its counts describe the calls the agent issued rather than being written by the enforcement path itself. Either way, the records yield the per category call counts and the number of denied and failed commands. Credential redaction is applied before text is printed or persisted to the paths through which injected credentials can echo back, namely the captured output of platform and interface subprocesses, the notes recorded for degraded steps, and the error column of the results table.

The metrics are tracked per run in `results/results.csv`. Table 3.2 lists its columns, which together record the complete identity of the run's cell, its outcome, its resource consumption, and the pointer to its artifacts.

Table 3.2: Columns of the results table `results/results.csv`, grouped by what they record. Every run appends one row.

Group	Columns
Cell identity	`config`, `model`, `platform`, `interface`, `version`, `skills`, `category`, `task`, `n`
Run metadata	`started_at`, `error`, `run_dir`
Outcome	`valid`, `success`

Table 3.2: Columns of the results table, continued.

Group	Columns
Assertions	asserts_passed, asserts_failed, asserts_skipped, total_asserts
Timing	wall_time_s, rate_limit_wait_s, platform_time_s, local_time_s, sleep_calls, sleep_time_s
Tokens and cost	input_tokens, output_tokens, total_tokens, cost_usd
Invocations	llm_calls, interface_calls, python_calls, bash_calls, read_calls, write_calls, edit_calls, glob_calls, grep_calls, todo_calls, skill_calls
Failures	failed_commands, denied_calls

Within the timing decomposition, wall_time_s already excludes rate limit waiting, which is reported separately in rate_limit_wait_s, and decomposes exactly into platform_time_s plus local_time_s. The results table reproduces every value reported in the results chapter and identifies the artifacts backing each row, namely the agent log, the per command record, and the grading report.

The remaining planning elements are the analysis procedure and the assessment of validity.

3.5.9 Analysis

The dependent variables are analyzed according to the rules of this subsection. Since each cell contains a single run, cells are not aggregated internally. Instead, results are aggregated across tasks, reporting per condition the success rate and mean assertion pass fraction as accuracy, and mean cost and turns, both overall and separately for feature, training, inference, and operations tasks and capstone projects. Invalid runs, in which no gradable deliverable was produced, are reported separately and scored as zero assertions passed, since producing a gradable deliverable is part of the task. Runs terminated at the one hour deadline retain zero accuracy, but their zero turn and local time fields are placeholders written when the process was killed rather than observed values, so those two fields are treated as missing in efficiency aggregates and paired tests.

Hypothesis testing uses the paired structure of the design. Every planned contrast of interest, command line interface (CLI) versus software development kit (SDK), skills versus none, one model versus another, one platform versus another, and each optimization versus its baselines, is evaluated over the common tasks available to both conditions. Runs terminated at the one hour deadline require a distinction between their metrics. Such a run produced no valid deliverable, so its zero accuracy is a true measurement and the run stays in the accuracy and cost contrasts. Its recorded zero turns and zero seconds, however, are placeholders written because the killed process never reported its final counts, while the run in truth consumed the full hour. Treating these zeros as measurements would make the most expensive runs look like the cheapest, so the turn and time values are treated as missing instead, which reduces the affected turn and time contrasts to 17 pairs. The reports therefore state both the paired sample size and, for signed-rank tests, the number of non-zero paired differences. Task instances are seeded from the treatment, so the interface and skills contrasts, which live inside one configuration, compare identical seeded instances, whereas the model, platform, and optimization contrasts compare different seeded instances of the same task template, so instance level difficulty variation enters those contrasts as noise rather than being controlled. The test is fixed per metric. The binary success indicator is compared with McNemar's test on the per-task agreement table [229], and the assertion pass fraction, cost in United States Dollar (USD), turns, and time are compared with the Wilcoxon signed-rank test, using the normal approximation with tie correction because the paired differences contain ties and zeros. The signed-rank test evaluates whether the distribution of paired differences is centered on zero under its symmetry assumption, and it is not interpreted as a distribution-free test of the arithmetic median. Zero differences are discarded by the test, so the effective non-zero sample is reported and rank-biserial values based on very few changes are interpreted cautiously. The paired t-test, which evaluates whether the mean of the task-wise differences differs from zero under an assumption of normally distributed differences [230], is reported alongside as a sensitivity check only. All tests are reported with effect sizes so that magnitude can be interpreted alongside statistical significance. Cohen's d is used for parametric tests as the mean of the paired differences divided by their standard deviation [231], and rank-biserial correlation is used for non-parametric tests to quantify the strength and direction of the non-zero ranked differences [232].

Because the design yields many contrasts, the analysis is restricted to defined contrast groups, one per manipulated factor. The interface contrasts

compare the CLI with the SDK within each model and platform, the skills contrasts compare skills with none within each interface, model, and platform, the model contrasts compare models pairwise within each platform and interface at matching skill conditions, the platform contrasts compare the European with the American platform within each model, interface, and skill condition, and the optimization contrasts compare each variant, the optimized skills, and their combinations with the baseline CLI and the SDK. Within each factor's contrast group, the p values of all inferential outcomes and, for optimization, both reference baselines are adjusted together for multiple comparisons with the Holm procedure [233], and any contrast outside these groups is reported as exploratory and descriptive. Conclusions for feature, training, inference, and operations tasks and for capstone projects, based on between two and seven tasks each, are likewise reported descriptively rather than tested. Because the 22 tasks are an authored, fixed population rather than a random sample from the space of machine learning (ML) platform tasks, statistical inference generalizes to this benchmark, and generalization beyond it is an argument from the benchmark's construction rather than from the tests. For RQ3, comparisons are made through the interface version column at fixed model, platform, and task, contrasting each variant with the baseline CLI and the native SDK on the same metrics, with particular attention to cost, turns, and platform time reductions at unchanged or improved accuracy. The treatment definitions are retained in the repository, but the analysis rules of this subsection, the test per metric, the contrast groups, and the adjustment procedure, were fixed in this chapter, but their implementation postdates the execution of the experiments, so the analysis is specified rather than preregistered, and this distinction is carried into the limitations. The final planning element assesses the validity of the conclusions this analysis can support.

3.5.10 Validity

The validity assessment covers the four validity dimensions, namely internal, external, construct, and conclusion validity, and names, for each threat, the design decisions of the meta-harness that mitigate it. It begins with internal validity.

3.5.10.1 Internal Validity

Internal validity is threatened by the possibility that differences in accuracy, cost, or turns reflect prompt sensitivity, stochastic variation in generation,

or platform-side transients rather than a genuine effect of the manipulated factors. Three design decisions of the meta-harness address this. The first is partial reproducibility through pinning. Interface builds, sets of skills, model identifiers, and task seeds are fixed before execution, so those inputs can be regenerated exactly, with the coding agent executables as the exception documented below. The second is isolation. Each run mints its own platform namespace, so concurrent and consecutive runs cannot observe or corrupt each other's state, and teardown is scoped to exactly that namespace. Every interface is built once into a prepared virtual environment that each run clones read-only using copy on write file cloning, so runs never reinstall dependencies or mutate shared state. The generated answer key is materialized outside the agent's file system boundary and additionally denied to the agent's tools by path. The third is enforcement over instruction. The restriction to a single interface is enforced at execution time by the allowlist rather than merely stated in the prompt. Further mitigations hold the prompt template, sandbox, enforcement policy, provisioning, and grading identical across conditions, with the sandbox asymmetries documented in the execution protocol and discussed below, verify platform readiness before and cleanliness after every run, and strip vendored agent instructions from interface sources so that no uncontrolled directives enter the context.

Several threats to internal validity remain and must be carried into the interpretation of the results. First, model and coding agent are confounded by construction, since each vendor's models run only through that vendor's coding agent, whose scaffolding, tool definitions, context management, retry behavior, and decoding parameters differ from the other coding agents and are outside the experimenter's control. Any difference attributed to model origin or tier is therefore strictly a difference between combinations of model and agent, and the results are reported and interpreted at that level. The enforcement mechanism is part of this confound in a weaker form, since the identical allowlist policy is applied through a tool use hook for Claude Code and through an interposed shell for the other coding agents, so the timing and shape of a denial are not guaranteed to be identical even though the policy and its logging are. The command sandbox layer described in the execution protocol deepens the same confound, since it exists only in the Claude coding agent, and within the Claude conditions it introduces one interface asymmetry on the American platform, where the platform's command line interface (CLI) binary is excluded from the sandbox because it cannot establish encrypted connections inside it, while the platform's software development kit (SDK) runs and all runs on the European platform execute

their commands inside it. The allowlist and the path denies apply to every call in every condition, so the enforcement of the interface restriction is unaffected, but the execution environment of that CLI is not identical to that of the other interface conditions. Removing both asymmetries would have required the uniform container-based sandboxing that the local execution setup did not provide and that future work names as the stronger design.

Second, because the coding agent versions were not pinned and the two platforms were executed in sequence, the Claude platform contrasts compare runs whose agent harness differs by several patch releases as well as by platform, so they are comparisons between the earlier runs on the European platform and the later runs on the American platform rather than comparisons under a frozen harness. The intervening changelog documents behavioral changes to the harness, but the data provide no overlap from which to estimate their effect. The platform effect is therefore inseparable from execution period and coding agent version.

Third, isolation guarantees state separation but not performance separation. Treatments executing in parallel on the European platform's dedicated cluster can contend for its resources, whereas the American platform absorbs contention invisibly as a managed service, an asymmetry that can enter the platform time measure and, through the budget, the outcome.

Fourth, with a single run per cell, any individual cell value carries sampling noise, which is addressed by drawing conclusions only from effects that are consistent across the 22 benchmark tasks rather than from individual cells.

Fifth, the run order was deterministic rather than randomized or counterbalanced. Within each task the interface and skill conditions therefore occurred in a fixed sequence, so short-lived changes in provider or platform behavior could align with condition order even though the isolated namespaces prevent state contamination. This is an additional reason to interpret the contrasts as observed associations rather than isolated causal effects. How far the results generalize beyond this setting is the concern of external validity.

3.5.10.2 External Validity

External validity is limited along five dimensions. First, the generated benchmark, while designed to span feature, training, inference, and operations tasks and capstone projects with realistic complications, is synthetic and may not capture the full messiness of production data and workflows, and since the tasks are an authored fixed population, statistical inference is to this benchmark rather than to machine learning (ML) platform tasks at large.

Second, although the meta-harness supports five platforms and three coding agents, the committed treatments cover only the two selected platforms with Claude and Mistral models due to financial restrictions, so cross platform conclusions are strongest for the observed contrast between the earlier European-platform runs and the later American-platform runs, and generalization to the remaining supported platforms, to medium and low coverage platforms, and to Open Artificial Intelligence (OpenAI) models under Codex is not directly supported.

Third, the optimization phase is conducted only on the European platform's CLI with a single model, so its findings establish feasibility rather than generality across platforms and models.

Fourth, the optimization artifacts were derived from runs covering the same authored templates later used for evaluation, while inference is reported on the 22 retained benchmark tasks, so no held out tasks test whether the optimized interface generalizes to unseen workloads. This restricts RQ3's answer to feasibility for a known workload.

Fifth, the 39 market candidates were assembled purposively from a vendor landscape, literature, a vendor-maintained directory, and news monitoring rather than through an exhaustive reproducible search with formal inclusion and stopping rules. Counts and uniqueness claims therefore apply to the assessed candidate set and should not be interpreted as complete population estimates of every platform in the market. Whether the measured quantities capture the intended constructs is the concern of construct validity.

3.5.10.3 Construct Validity

Construct validity depends on whether the grading adequately operationalizes task success, on whether the benchmark is a platform neutral and model neutral instrument, and on whether the recorded counts adequately operationalize turns and cost. The meta-harness grades by state, not by transcript. A task counts as solved only if the required artifact exists on the platform and passes its assertion suite when read back through the platform's own data paths by the checker adapter, so a plausible transcript or a local file contributes nothing. This measures the state the agent actually produced, which is the outcome of interest for platform operation, makes grading robust against superficially convincing agent output, and turns evaluation into a state verification problem, following the direction established for branching lakehouses [221]. However, its accuracy depends on the correctness of the checker adapters, which are exercised by the meta-harness's test suite and by the generators' self tests

during preparation, and on the single attempt read policy. That policy carries a residual false negative risk. Because a deliverable read is attempted exactly once, a brief platform outage at the moment the grader reads can mark a deliverable as missing even though it exists. The reliability probe during preparation established that such transients are rare on a healthy service, the preserved failure reasons make them identifiable during data validation, and the one transient query service timeout observed among the committed runs was re executed under the retry rule, so no retained grade is knowingly affected. The residual risk is accepted as the price of grading through the platform's own read path rather than the agent's transcript, which is what forces the agent to interact with the platform at all, at the cost of some flexibility in how the grading can respond to platform conditions.

The benchmark is organized around the feature, training and inference (FTI) architecture defined by Dr. Jim Dowling, the founder of the European platform's vendor, the capstone projects derive from that vendor's affiliated `mlfs-book` example systems, and the tasks were generated with a Claude model while Claude models are among the evaluated subjects. Against the platform-side of this concern, the FTI decomposition is the field's organizing abstraction for ML systems rather than a proprietary one, the same pipeline stages structure the documentation of all high coverage platforms including the American platform, the task prompts are phrased in domain language and translated to each platform by its own adapter, and grading by state is symmetric across platforms. Against the model side, the validity checks constrain every task to a computable ground truth independent of any model, and the cross-vendor review pass checked the tasks with a Mistral agent. What these mitigations cannot exclude is a residual distributional proximity, in phrasing, naming, and solution idiom, between the tasks and the priors of the model class that generated them, since the review pass detects defects rather than bias, and the public availability of the `mlfs-book` pipelines means the capstone structure is plausibly represented in the training data of all evaluated models, which fresh seeds mitigate only at the level of data and resource names. Results on the platform origin contrast and on the capstone projects are therefore interpreted with these caveats attached.

Cost is operationalized for Claude as the cache-inclusive billed cost derived from the retained session transcripts, as described in the instrumentation, while Mistral costs are lower bounds because cache usage is not retained, so cost comparisons are made between Claude cells only. The United States Dollar (USD) figure additionally operationalizes vendor pricing as much as computational effort, a caveat attached wherever cost is interpreted, and

token counts are recorded descriptively alongside. It covers model invocation charges only and excludes platform compute, storage, warehouses, clusters, and networking, so it is not a total cost of ownership or an end-to-end workflow cost. Turns are measured as model invocations and decomposed into per category tool calls counted by the same logic that enforces the allowlist, so interface calls cannot diverge from what the agent was actually permitted to execute. Because coding agents decompose work into invocations differently, turn comparisons are well defined within a coding agent, across interfaces, skills, and optimization conditions, and only indicative across coding agents. Tool call counts do not distinguish substantive interface work from auxiliary calls such as file reads, which is taken into account when interpreting turn differences between interfaces. Latency is decomposed so that platform-side latencies and provider rate limit waiting, which reflect neither the interface nor the agent, are subtracted from the interface latency measures, so the resulting local time is not end-to-end user waiting time. Deadline-terminated processes do not emit final turn or timing metadata, so their zero placeholders are treated as missing for efficiency analysis rather than as zero-resource observations.

The experiment operationalizes model tier through vendor product designations, such as Opus versus Sonnet and Large versus Medium. Proprietary parameter counts and training compute are not observed and are not assumed comparable across-vendors, so tier findings concern the evaluated product variants and combinations of model and agent rather than measured parameter count or training scale.

3.5.10.4 Conclusion Validity

Conclusion validity depends on the paired design, in which tests are applied to paired per-task differences over the common tasks available to both conditions, and effect sizes are reported alongside significance. Missing final metadata from deadline-terminated runs reduces affected turn and time contrasts to 17 pairs. The paired and non-zero sample sizes are reported alongside every test. The latter matters because the Wilcoxon procedure discards zero differences, so a rank-biserial value near one can come from very few changed tasks and is interpreted together with its effective sample.

The principal limitation is that, with a single run per cell, within cell variance attributable to large language model (LLM) non-determinism cannot be estimated, so a paired difference on any individual task is compatible with run noise, and only between condition differences that are consistent across tasks and, where applicable, across platforms and models provide

evidence, while differences observed for individual tasks or the five named task categories are interpreted descriptively and with caution.

A second limitation is that the tasks are heterogeneous and commonly authored rather than exchangeable random draws. The signed-rank symmetry assumption is not guaranteed by this construction, so the paired t-test is retained as a sensitivity check and inference remains restricted to the benchmark. The multiplicity of contrasts is handled by adjusting all inferential outcomes within each manipulated factor's contrast group together with the Holm procedure. The figures and statistics reported in the results chapter are produced by the versioned analysis scripts stored alongside the figures in the repository, which constitute the authoritative analysis pipeline, while the exploratory notebook is retained for transparency but is not load bearing. Conclusion validity is further supported by full provenance in a single table. Every run appends exactly one row to the results comma-separated values (CSV), carrying the complete identity of its cell, including the interface version string and skills label, its outcome, and a pointer to its artifact directory, so that any reported number can be traced to the exact run, prompt, command sequence, and grading report that produced it, and re-running a treatment never overwrites, since completed cells are skipped and retries accumulate as new repetition indices. The retained treatment definitions document the intended design, while the analysis implementation postdates execution as stated above. The single model scope of the optimization phase limits the strength of any inference specific to those optimizations.

With the experiments fully planned, the next phase concerns executing the plan by operating the experiments.

3.6 Operating the Experiments

The operation phase consists of three activities. Preparation readies the meta-harness, the benchmark, and the platforms, execution carries out the committed treatments, and data validation checks the collected data for correctness and completeness before analysis. The first of these activities is preparation.

3.6.1 Preparation

Preparation covers the work of readying the meta-harness, the benchmark, and the platforms before any committed treatment was executed. It comprises implementing the `mlpab` Python package and its unit test suite,

generating and verifying the benchmark, bootstrapping the platform accounts, building the pinned interface artifacts, and verifying the meta-harness end-to-end through the preflight checks. Once preflight passed for every committed treatment, execution began.

3.6.2 Execution

The runs were executed between 17 June and 4 July 2026 by starting each treatment through the meta-harness. The four comparison phase grids on the European platform ran from 17 to 20 June, the optimization artifacts were then produced from their run records, the thirteen evaluation phase treatments ran from 22 to 24 June, and the four grids on the American platform ran from 29 June to 4 July. The chronological order therefore interleaved the logical phases. The interface optimization required only the European platform's command line interface (CLI) run artifacts of the comparison phase, so the remaining treatments could execute after the evaluation phase without affecting the design, since every treatment was independent of every other and no optimization artifact depended on the later treatments.

The committed design comprised 990 planned cells, 704 in the eight comparison grids and 286 in the thirteen evaluation grids.

In aggregate, the retained runs consumed 175 hours of agent compute time and a further 96 hours of rate limit waiting, which the budget and the recorded wall times exclude, 37.2 million fresh input and output tokens plus a substantially larger volume of separately billed cache traffic, at a total model cost above 1,082 United States Dollar (USD), of which the Claude runs account for 1,027 USD under the cache-inclusive accounting described in the instrumentation and the Mistral figures remain lower bounds, 44,445 model invocations, and 30,016 interface calls. The enforcement recorded 1,191 denied commands and 1,277 failed commands across the runs, confirming that the allowlist was exercised in practice rather than remaining a dormant safeguard, and 11 runs were terminated by the meta-harness at the deadline. Nine runs exceeded the nominal one hour budget by up to 31 minutes. In each case the recorded timing decomposition showed the excess inside platform-side work, long running platform jobs or agent initiated waits on them, because the deadline did not kill an in flight interface call, and all nine runs completed with a gradable deliverable and were retained. The final activity validated the data these runs produced.

3.6.3 Data Validation

The collected data was validated before analysis by checking the results table for correctness and internal consistency and by applying two data reduction rules fixed in advance of the analysis.

Two predetermined reductions were applied: rows outside the committed design were excluded, and for the only repeated cell the latest attempt was retained. The analysis set therefore contained 986 unique cells, comprising 700 comparison rows and 286 optimization rows. Four planned comparison observations were unavailable and excluded from aggregates, figures, and tests. Incomplete bars show their retained and planned sample sizes.

The analyzed rows were matched to their committed treatment definitions. Every analyzed row belonged to a committed treatment and no cell identity occurred twice. Correctness was verified through the internal identities of the schema, each holding on every analyzed row without exception. Success implies validity, the passed, failed, and skipped assertion counts sum to the assertion total, the wall time decomposes exactly into platform time plus local time, and input and output tokens sum to the total token count. The version column confirmed that every run executed against its pinned build, with exactly one version string per platform and interface pair, `5.1+alerting` and `5.0.0` on the European platform and `1.3.0` and `0.117.0` on the American platform, and one distinct `5.1+opt` string per command line interface (CLI) variant.

Of the 986 retained rows, 850 were valid and 136 were invalid, and every invalid row carried a machine recorded reason in its error field.

MLPlatformAgentBench (MLPAB) ran on a 2021 M1 Max MacBook Pro with 32 gigabyte (GB) of memory, macOperating system (OS) 26.5.1, Python 3.13.13, and GNU Make 3.81. The evaluated platforms run remotely, and the meta-harness supports five platforms, including the two evaluated here.

With preparation, execution, and validation complete, the next chapter reports the results and analysis.

Chapter 4

Results and Analysis

This chapter carries out the fourth phase of the experiment process, analysis and interpretation, on the outputs of the five method phases. It begins with the market assessment, whose coverage matrix answers RQ1 and identifies the platforms on which the experiments were conducted, followed by the benchmark results and the optimizations evaluation.

4.1 Market Assessment

The market assessment scored 39 candidate machine learning (ML) platforms against the criteria defined in the method chapter, mapping the agent-facing interfaces and agents, the Python software development kit (SDK), the command line interface (CLI), the Model Context Protocol (MCP) server, and ML platform agents, onto the four automated machine learning (AutoML) stages of the MLPlatformAgentBench (MLPAB) benchmark, namely feature, training, inference, and operations. The score extends the MCP class with documented server hosting and the ML platform agent class with the PySpark, file system, and dashboard capabilities available to the agent. From these criteria, the market coverage score aggregates 23 coverage criteria: four stages each for the Python SDK and the CLI, five for the MCP including server hosting, seven for ML platform agents including their three additional capabilities, one for a documented meta-harness, the second route by which a coding agent operates a platform, one for a published set of skills, and one for documented agent deployments. A fully supported criterion contributes one point and a partially supported criterion half a point. Active maintenance, a public software as a service (SaaS) offering, headquarters location, an agent that meets neither agent definition, and the four entries recording whether each

interface exists and where it is documented do not enter the score. Hopsworks holds no ML platform agent under this definition. What its vendor publishes is a meta-harness, the Hops Terminal, through which general-purpose coding agents operate the platform, and it is scored as such.*

The central finding of the assessment, summarized in Figure 4.1, is twofold. First, the MCP remains underdeveloped compared with the CLI and the SDK. Of the 39 assessed platforms, 29 document a Python SDK, 27 a CLI, 18 an MCP server, and four document an ML platform agent in the sense defined in the method, while five platforms cover all four stages through their SDK, three through their CLI, and two through ML platform agents, while not a single MCP server does. A further twelve vendors ship an agent that does not meet that definition, a software agent for analytics or office work such as Microsoft 365 Copilot, Google's conversational analytics, or Snowflake CoWork [234], an assistant that suggests rather than executes such as the Statistical Analysis System (SAS) Viya Copilot, or a pipeline agent confined to another domain such as Astronomer's Otto. These are recorded in the assessment but score nothing, because they do not carry out feature, training, or inference workloads on the platform. The stage decomposition of Figure 4.1 sharpens this contrast. Through the SDK, training is the best-covered stage, reached by 50% of the market, whereas through MCP tools and ML platform agents, training is the weakest stage, reached by 15% and 6% respectively. The MCP and ML platform agents, the newer of the four, therefore expose retrieval and operations far better than they expose the model-building core of the lifecycle. The strongest MCP servers, those of Teradata, Databricks, SAS, and Cloudera, still fall short of covering even a majority of the stages in full. Second, no platform fully covers the entire spectrum that the benchmark operationalizes, namely the feature, training, and inference pipelines extended by large language model (LLM) fine-tuning and serving, together with machine learning operations (MLOps), across the SDK, the CLI, the MCP, and ML platform agents. Even Databricks, the highest-scoring platform, covers this spectrum in full only through its SDK, CLI, and ML platform agents, while its MCP server does not, and every other platform leaves larger gaps. Two further findings complete the picture. The supporting ecosystem for agent operation is thinner still. Only eight vendors publish skills, nine document hosting for MCP servers, and seventeen document deployments of an organization's own agent in the platform. These supporting artifacts and deployment paths therefore remain minority features. The market is also geographically concentrated. Of the 39 vendors, 31 are

*The complete assessment matrix is reproduced in Appendix B.

headquartered in the United States, six in Europe, of which three are in Germany, two in Sweden, and one in Switzerland, and one in China, while one is a distributed open-source project. Iguazio is counted under the United States instead of Israel following its acquisition by McKinsey & Company. Only a single European platform reaches the leading group, a finding directly relevant to the sovereignty dimension of RQ2. Of the 39 platforms, 32 are actively maintained and 20 are offered as a public SaaS.

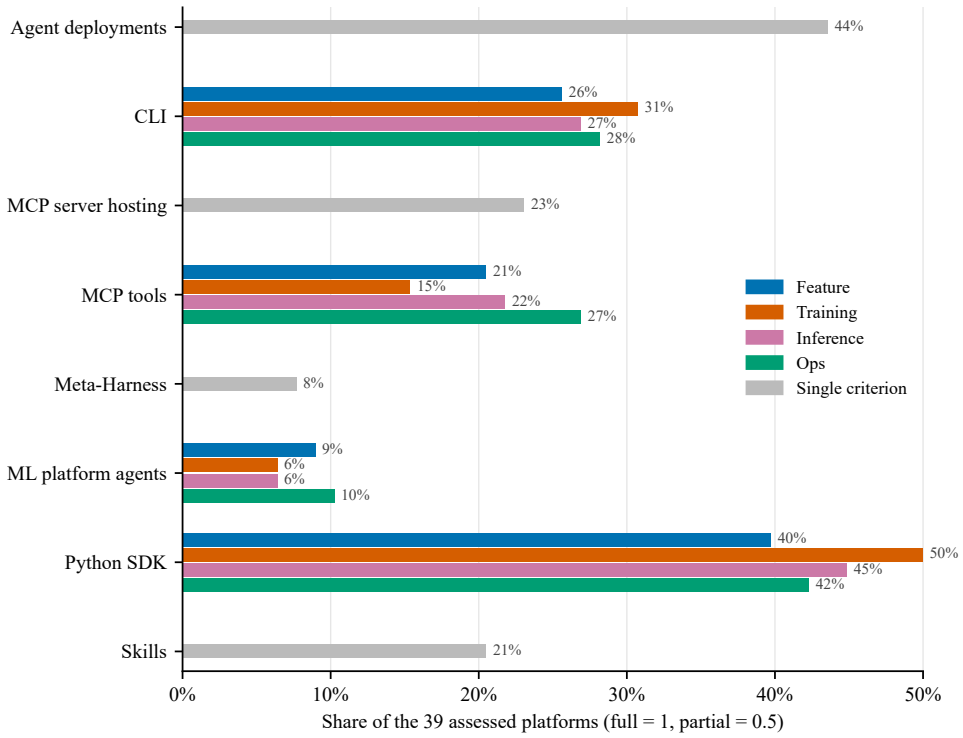

Figure 4.1: Share of the 39 assessed platforms covering each AutoML stage through the CLI, MCP, ML platform agents, and SDK, with full support scoring one and partial support half. Agent deployments, MCP server hosting, meta-harness, and skills are single criteria shown as gray bars. The axis is capped at 50%, the next step above the largest share, for readability.

Figure 4.2 presents the five leading platforms and Figure 4.3 the remaining 34, with each platform's market coverage decomposed into the seven scored interaction groups. The adjacent matrix shows active maintenance, headquarters, public SaaS offering, and an agent that does not meet the ML platform agent definition. None of these four facts enters the score, and every scored criterion appears only in the bars, so nothing is counted twice across

the two panels. The leading group is defined as the actively maintained, publicly available platforms scoring at least 35%, a rule applied uniformly to all 39 platforms on facts the matrix already records, so that a platform an organization cannot adopt today does not lead the market. It excludes Teradata at 39% and Iguazio at 33%, neither of which is offered as a public SaaS, the latter following its acquisition. The leading five span 43% to 93%.

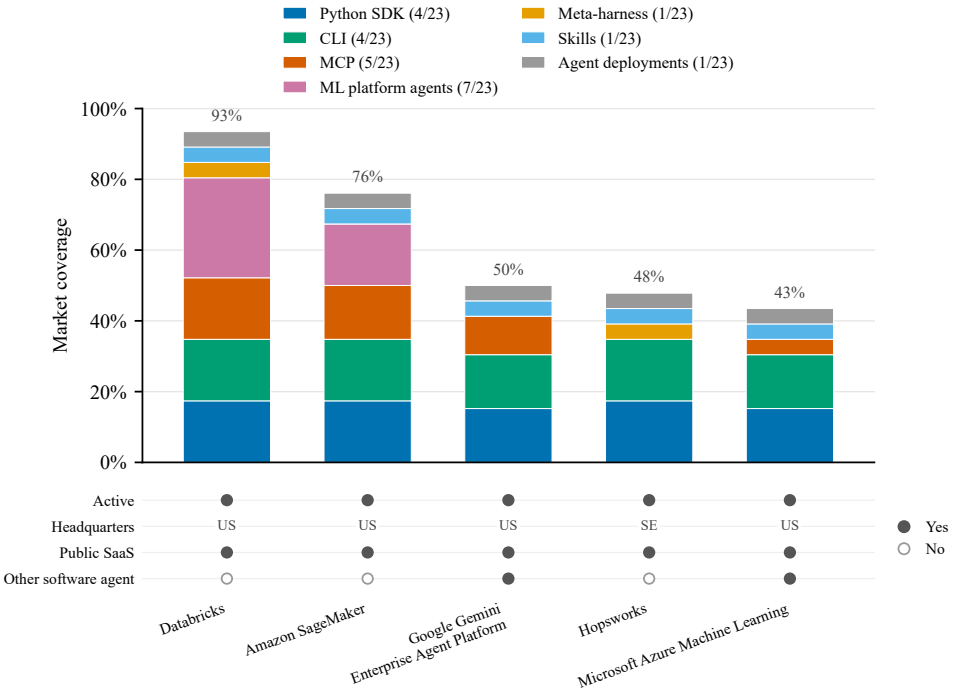

Figure 4.2: Market coverage of the five leading platforms, decomposed into the Python SDK and CLI groups of four stages each, the MCP group of four stages plus documented server hosting, the ML platform agent group of four stages plus the PySpark, file system, and dashboard capabilities available to the agent, the documented meta-harness, the published skills, and the documented agent deployments as single criteria each, out of 23 criteria points in total. Headquarters are abbreviated as US for the United States, DE for Germany, SE for Sweden, CH for Switzerland, and CN for China, and a dash marks a distributed open-source project without a corporate seat.

The five leading platforms, shown in Figure 4.2, score between 43% and 93%.

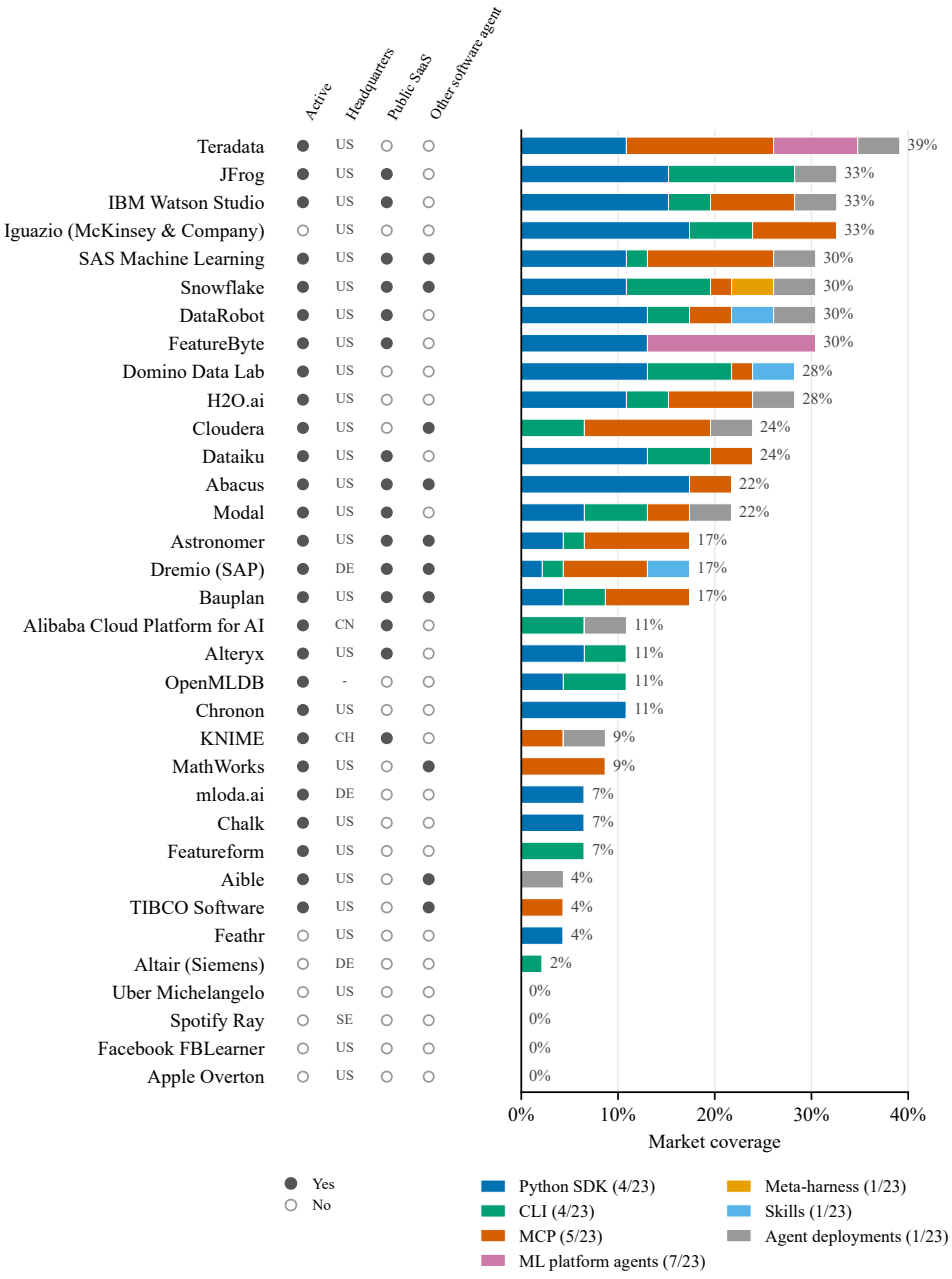

Figure 4.3: Market coverage of the 34 platforms outside the leading group, decomposed as in Figure 4.2. Acquired platforms carry their acquirer in brackets. The axis is capped at 40%, the next step above the strongest score in the group, for readability, and platforms with equal scores are ordered by their number of satisfied facts. The four platforms at zero either expose no public interfaces to assess or satisfy none of the scored interface criteria.

All five are actively maintained public SaaS offerings, all five achieve full or near full coverage of all four stages through both their Python SDK and their CLI, all five publish skills, and all five document deployments of an organization's own agent. They differ primarily in the maturity of their MCP servers and in whether the vendor ships an agent that operates the platform at all, which only Databricks and Amazon SageMaker do, while Hopsworks instead publishes a meta-harness and Microsoft and Google ship agents aimed at office work and at conversational analytics. Four of the five are American, and the sole European entrant is Hopsworks.

The remaining 34 platforms, shown in Figure 4.3, span scores from 39% down to zero, and their coverage follows three recurring patterns. First, coverage is centered on the SDK. The Python SDK is the strongest interface for the majority of the group, only Iguazio and Abacus reach full coverage of all four stages through their SDK, and no platform in the group does so through its CLI, whose command surfaces are predominantly administrative or infrastructure oriented rather than ML lifecycle interfaces, exemplified by Teradata and Abacus, whose otherwise substantial coverage includes no ML capable CLI at all. Second, MCP servers are common but partial. Thirteen platforms in the group publish one, and for six of them, Teradata, SAS, Cloudera, Astronomer, Bauplan, and Dremio, the MCP server is the strongest interface the vendor offers, inverting the pattern of the leading group, while MathWorks offers an MCP server as its only assessed agent operable interface, yet in no case does it approach full coverage of the stages. Six platforms in the group, Abacus, DataRobot, Dataiku, Modal, Konstanz Information Miner (KNIME), and The Information Bus Company (TIBCO) Software, document MCP server hosting without publishing a server of their own. Third, the supporting ecosystem is largely absent, with only three vendors in the group, Dremio, Domino Data Lab, and DataRobot, publishing skills, and none documenting the combination of full coverage interfaces and skills that the experiments of this thesis require. Teradata tops the group at 39%, followed at 33% by Iguazio, whose acquisition by McKinsey & Company has ended its availability as an active public offering, by International Business Machines (IBM) Watson Studio, and by JFrog, whose ML product covers the feature, training, and inference stages through its Python SDK atop a platform otherwise oriented to artifact management, and then by FeatureByte at 30%, the latter notable for ML platform agents assessed at full coverage of all four stages atop a platform that otherwise offers only an SDK. Bauplan, whose skill optimization study is the closest related work to RQ3 [221], sits mid group at 17%, its agent first design notwithstanding, because its scope is

data engineering rather than the MLOps and AutoML tasks the benchmark operationalizes, so the training stage remains uncovered across all of its interfaces and inference is reached only partially, through its MCP server. The lower half of the group holds platforms with fragmentary interfaces, typically a single partially ML capable SDK or CLI, including the feature store specialists Chronon, Featureform, Chalk, and Feathr, the open source Open Machine Learning Database (OpenMLDB), and Alibaba Cloud Platform for artificial intelligence (AI), the sole Chinese vendor in the assessment, whose public English language documentation supported credit only for a partially ML capable CLI and a documented agent deployment. At the bottom of the group, Aible and TIBCO Software score a single criterion each, a documented agent deployment and documented MCP server hosting respectively, and satisfy none of the stage coverage criteria. Four platforms score zero, the proprietary internal platforms Apple Overton, Facebook Facebook (FB)Learner, Spotify Ray, and Uber Michelangelo, which expose no public interfaces to assess and are retained in the matrix only to document the historical landscape from which the commercial market emerged. The group also sharpens the sovereignty finding of the assessment. Of the six European platforms assessed, five fall outside the leading group, with Dremio, now part of System Applications and Products in Data Processing (SAP), the strongest among them at 17%, so Hopsworks is the only such option within the assessed candidate set at the assessment date.

In answer to RQ1, the assessed candidate set contains 39 identifiable platforms, of which 32 are actively maintained, but agent operable coverage of MLOps and AutoML tasks is concentrated in five leading platforms dominated by American hyperscalers and platform vendors. The SDK remains the most complete interface and the CLI reaches full coverage of all four stages on exactly three platforms, Databricks, Hopsworks, and Amazon SageMaker, whereas the MCP remains underdeveloped relative to both, reaching full coverage on no platform at all, and skills exist for only eight vendors. Consequently, no assessed platform fully covers the spectrum of feature, training, and inference tasks extended by LLM fine-tuning and serving, together with MLOps, across the SDK, the CLI, the MCP, and ML platform agents. Full coverage of this spectrum through both a CLI and an SDK, the precondition for the interface comparison of this thesis, exists on precisely three platforms, and on exactly one platform of European origin.

The five leading platforms are also the five platforms supported by the MLPAB meta-harness, each with its own platform definition and checker adapter, so the benchmark can be executed against any of them. Of these, the

committed experiments run on Databricks, the highest scoring platform, and Hopsworks, the highest scoring European platform and fourth overall, the pair selected by the method's rule of taking the highest scoring platform overall and the highest scoring European platform. Both provide a CLI and a Python SDK with full coverage of the feature, training, and inference pipelines as well as operations, so the interface comparison of RQ2 measures interaction style rather than coverage gaps, both publish skills, so the skills condition exists on both platforms, and they contrast an American with a European vendor, so the platform origin dimension of RQ2 can be studied without confounding origin with interface coverage. Amazon SageMaker, the only other platform with full CLI and SDK coverage of all four stages, is American, and no European platform besides Hopsworks reaches even the upper half of the remaining group. Hopsworks therefore enters the experiments as the European platform and Databricks as the American platform, with the interface version strings recorded in the results table, `5.1+alerting` and `5.0.0` on Hopsworks and `1.3.0` and `0.117.0` on Databricks, identifying the pinned CLI and SDK builds under test. The remainder of this section introduces the two platforms selected for this study, the European platform Hopsworks and the American platform Databricks, in detail.

4.1.1 Hopsworks

Hopsworks, headquartered in Sweden, scores 48%, the fourth highest of the assessment, and is the only non American platform in the leading group, indeed the only European platform in the entire assessment with more than fragmentary agent operable coverage of the benchmark stages. The platform is built around its feature store and the feature, training and inference (FTI) pipeline architecture of feature, training, and inference pipelines introduced in the background chapter [37, 36], and its interface surface is notable for being consolidated in a single open source repository. The Python software development kit (SDK) [235], the command line interface (CLI) [236], the Model Context Protocol (MCP) server [237], and the skills [238] are all developed in the `hopsworks-api` source tree. Both the SDK and the CLI achieve full coverage of all four stages, allowing an agent to register feature groups, run materialization and validation, build point in time correct training datasets through feature views, execute training jobs, register models, deploy online and batch inference, and manage jobs, schedules, and alerts, the last capability completed on the CLI by the alerting command group contributed upstream as part of the capability

parity correction described in the method chapter.

The MCP server, by contrast, was present in the source tree at assessment time but documented only login, project management, and feature-store handle-retrieval tools, so no stage coverage was credited to it, and no hosting for MCP servers is documented. Hopsworks does not supply a vendor-specific platform agent. It satisfies the meta-harness criterion through the Hops Terminal, which hosts different coding agents, such as Claude Code, within the platform itself, presented publicly by the vendor [239]. The meta-harness scores as a single criterion rather than as stage coverage, because it supplies no capability of its own. What it admits is an arbitrary coding agent operating the platform through the same full-coverage CLI and SDK, which the assessment already scores in those classes. The platform's agent deployment guides document the corresponding agent deployments fact [240]. Its combination of a full-coverage CLI and SDK, skills, and European origin makes it the natural counterpart to Databricks, and it enters the experiments as the European platform. The following subsection turns to that counterpart.

4.1.2 Databricks

Databricks achieves the highest coverage score of the assessment at 93%, and is the only platform to reach full or near full stage coverage across all four of these interfaces and agents. The platform is organized around the lakehouse architecture, unifying data engineering, machine learning (ML), and analytics workloads over shared governed storage, and exposes the full ML lifecycle to agents through two complementary programmatic interfaces. Its Python software development kit (SDK), comprising the general purpose Databricks SDK for Python [241] together with the dedicated feature engineering package for its feature store [242], allows an agent to create and populate feature tables, assemble training datasets, launch and monitor training jobs, register models, deploy and query serving endpoints, and manage schedules and alerts, covering all four benchmark stages in full. Its command line interface (CLI) [243] matches this coverage stage for stage, spanning workspace, jobs, serving endpoint, and quality monitoring command groups, which makes Databricks one of only three platforms whose CLI alone can carry the entire MLPlatformAgentBench (MLPAB) benchmark.

The agent-native surface is equally developed. The Model Context Protocol (MCP) interface, provided through the artificial intelligence (AI) Development (dev) Kit [244], fully covers inference and operations but only partially covers the feature and training stages, and Databricks is additionally

one of the few vendors to document hosting for custom MCP servers, deployed as apps on the platform itself [245]. Databricks is the only platform to satisfy both agent criteria. Its ML platform agent, Genie Code [246], is assessed at full coverage across all four stages, and its meta-harness, Omnigent, wraps any agent, including Claude Code and custom agents, in a sandboxed session with a uniform application programming interface (API), with a server that provides policies and sharing and exposes every session over the terminal, the app, and web APIs [171, 172]. Its documented agent framework additionally provides infrastructure for organizations to build and launch their own agents on the platform [245]. A repository of more than thirty skills in the agent-skills specification format [247] supplies the skills condition on this platform in the experiments. Among the additional ML platform agent capabilities, PySpark support and dashboards are fully supported, while file system access is assessed as partial because agents interact with workspace files and volumes rather than a general-purpose file system. Databricks therefore presents the most complete agent-facing surface in the assessed market, consistent with its position as a leader in the vendor landscape reviewed in the related-work chapter [181], and it enters the experiments as the American platform.

With the market assessed and both platforms introduced, RQ1 is answered, and the next section reports the results of the comparison phase, in which the coding agents operate both platforms across interfaces and skill conditions.

4.2 Benchmark

The benchmark comparison answers RQ2. The retained table contains 700 comparison rows. Of these, 174 nominal Mistral skills rows are retained for provenance but excluded from comparative figures and tests because the treatment was not delivered, leaving 526 valid treatment comparison rows. The 286 optimization rows are analyzed with the optimization results. Figures 4.4 and 4.5 show the mean assertion pass fraction per model, interface, and skills condition on each platform.

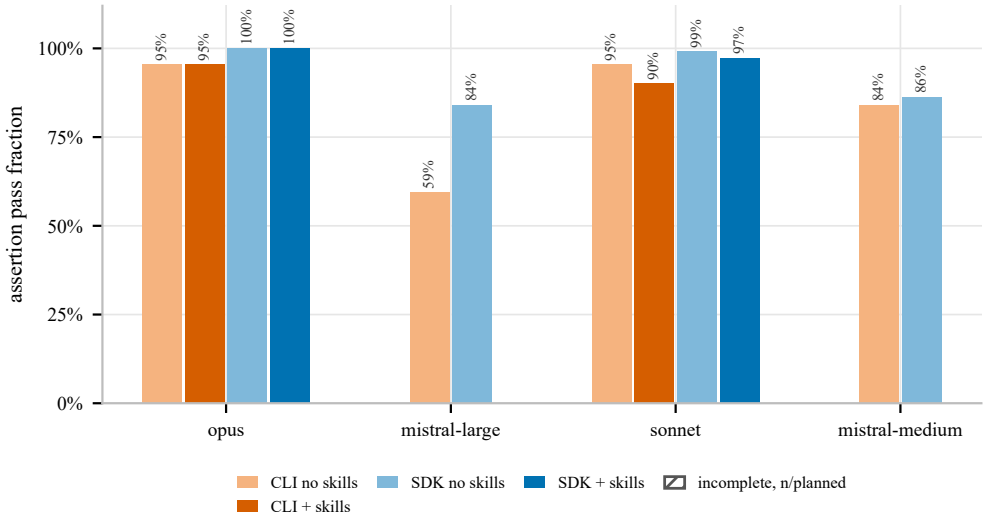

Figure 4.4: Mean assertion pass fraction per model, interface, and skills condition on Hopsworks. Invalid runs are scored as zero. Gray encodes the command line interface (CLI) and blue the software development kit (SDK), light shades the none condition and dark shades the skills condition, which was delivered to the Claude agent only, so the Mistral models appear without a skills condition. Hatched bars are incomplete and report their retained and planned sample sizes. A cross at the baseline marks an incomplete bar whose observed value is zero.

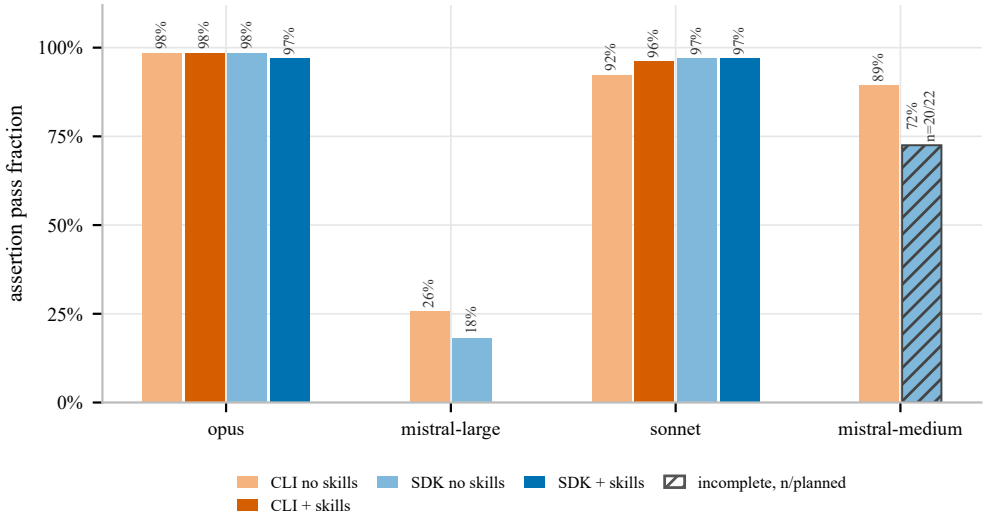

Figure 4.5: Mean assertion pass fraction per model, interface, and skills condition on Databricks, encoded as in Figure 4.4.

The dominant factor in Figures 4.4 and 4.5 is the model. Claude Opus passes 98% of assertions and solves 98% of its runs completely, Claude Sonnet passes 96% and solves 92%, Mistral Medium passes 83% and solves 80%, and Mistral Large passes 47% and solves 36%. The aggregate of Mistral Large conceals a platform split rather than a uniform weakness, as it passes 72% of assertions on Hopsworks but 22% on Databricks, where most of its SDK runs fail to produce a valid deliverable. Every other model performs nearly identically on the two platforms, with per platform pass fractions within two percentage points of each other. Within each model, the interface and skill variants lie close together, which anticipates the outcome of the planned contrasts.

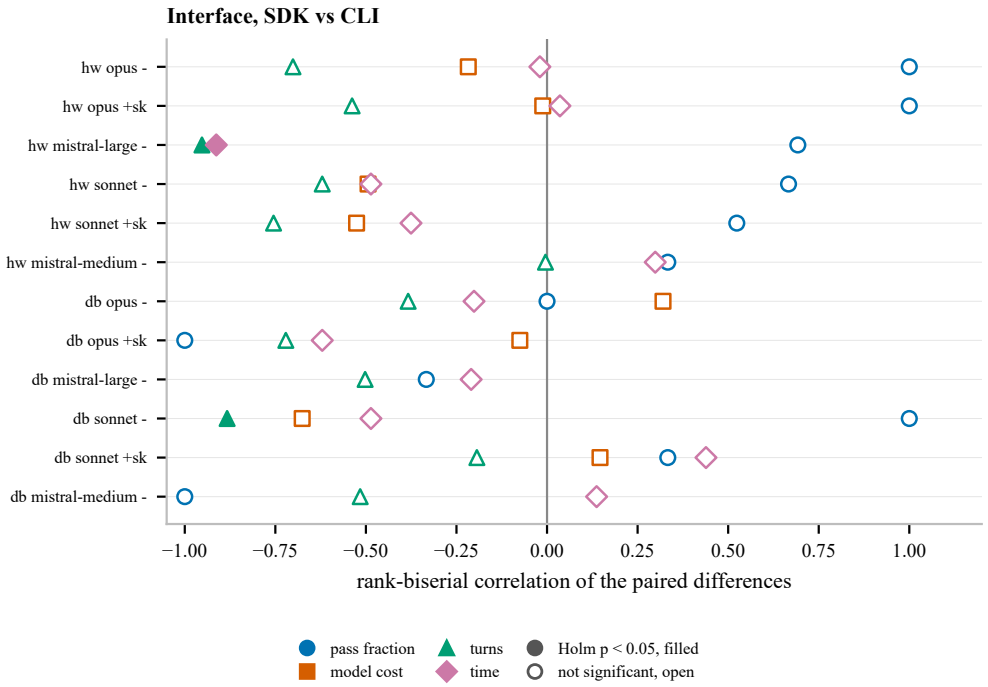

Figure 4.6: The interface contrasts of RQ2, stratified by skills condition. Each row is one paired Wilcoxon contrast over the benchmark, each dot the rank biserial correlation of the paired differences for one metric, filled when significant after Holm adjustment across all inferential outcomes of the interface factor. Positive values mean the SDK has the higher value, and rows marked `+sk` are the skills stratum.

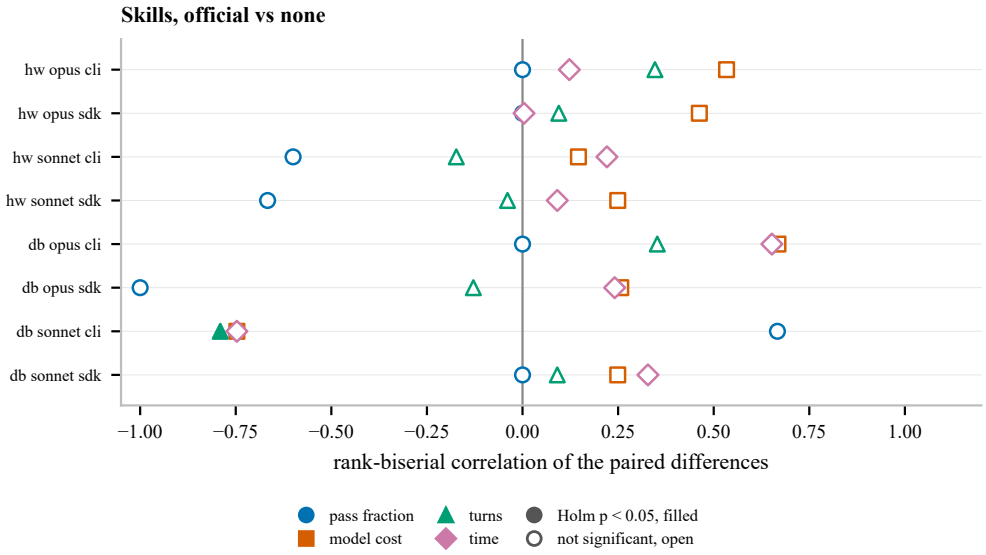

Figure 4.7: The skills contrasts, decomposed as in Figure 4.6, restricted to the Claude cells because the skills treatment was not delivered to the Mistral agent. Positive values mean the skills condition has the higher value.

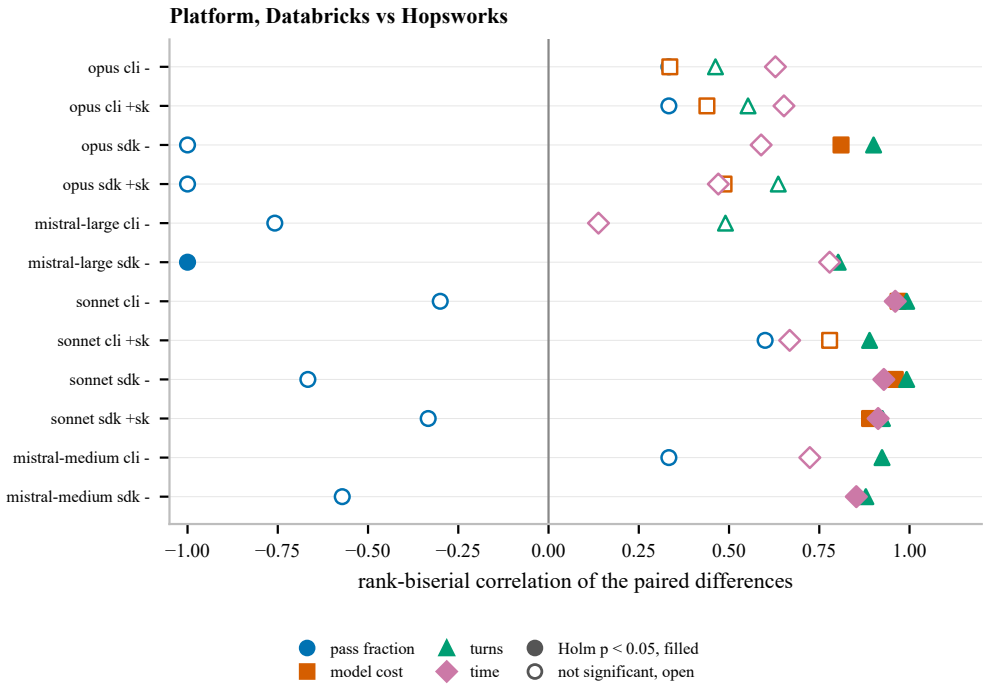

Figure 4.8: The platform contrasts, decomposed as in Figure 4.6. Positive values mean the later Databricks runs have the higher value, and rows marked `+sk` are the skills stratum. Because coding agent version moved with platform, the contrast does not isolate a platform effect.

The interface contrasts did not reject the planned accuracy null hypotheses. Stratified by skill condition to avoid pooling correlated observations, none of the twelve contrasts in Figure 4.6 is significant after factor-wide Holm adjustment on the assertion pass fraction, and McNemar's test on the per-task success indicator likewise finds no significant contrast. The Claude models contribute both skill strata while the Mistral models contribute only the no skills stratum, since the sets of skills never reached their agent, as the skills analysis below establishes. Aggregated over the analyzed cells, the SDK reaches a mean pass fraction of 87% against 85% for the CLI, a descriptive difference that no test supports. The efficiency metrics separate the interfaces on turns. The SDK completes tasks in significantly fewer turns in two of the twelve contrasts, with eleven of the twelve effects pointing in that direction, and time separates the interfaces in one contrast. Cost inference is restricted to the eight Claude contrasts, since the Mistral runner did not retain the separately billed cache usage, and none remains significant after the factor-wide adjustment. Descriptively, the mean Claude run costs 1.87 United States Dollar (USD) through the CLI against 1.51 USD through the

SDK, since every CLI turn rereads the growing cached context at the cache read rate, so many small turns are not cheap once cache traffic is billed. The paired t-test reported as a sensitivity check agrees with the rank based tests in direction in eleven of twelve contrasts on turns and on time, and seven of eight on cost. The directional expectation stated in the hypotheses, that the progressive disclosure of the CLI reduces model invocation cost and turns relative to the SDK, is therefore supported for neither metric, reversed for turns and unsupported for cost. The CLI agent works in many small steps, issuing one command per turn and reading its output, while the agent using SDKs batches many operations into fewer but heavier script executions.

The skills analysis carries a delivery caveat established by the post execution audit. The skills were installed at the Claude coding agent's discovery location, the Mistral session logs list no platform skill, and the Mistral skills rows record zero skill invocations against 140 skill invocations in the Claude skills rows, so the skills condition was a no op for the Mistral agent. Those rows are therefore excluded from every figure and contrast group, Mistral appears throughout with its no skills condition only, and the skills contrast group is restricted to the eight Claude contrasts in Figure 4.7. Within these contrasts, no statistically detectable difference in accuracy was found between the skills and none conditions. None of the eight contrasts is significant after factor-wide Holm adjustment on the pass fraction or under McNemar's test on the success indicator, and the descriptive Claude aggregate is 96% with skills against 97% without them. Only one efficiency contrast remains significant across the group. Claude Sonnet on the Databricks CLI needs fewer turns with skills. No cost or time contrast survives the factor-wide correction. On these tasks, no statistically detectable difference in accuracy was found for the Claude models in either direction, which is a statement about detectability rather than established equivalence, and the data support no statement at all about skills for the Mistral models.

The platform contrasts compare the two platform configurations and separate their observed efficiency from accuracy. Aggregate accuracy is 91% on the earlier Hopsworks runs against 82% on the later Databricks runs, but Figure 4.8 shows that the gap is carried entirely by Mistral Large, whose SDK contrast is the only significant accuracy contrast in the group, with a rank biserial correlation of one, and the only McNemar significant one, with thirteen tasks flipping against Databricks and none the other way. For the other three models no difference in accuracy is statistically detectable. The efficiency picture is one sided. The Databricks runs are significantly more expensive in four of the eight Claude model-invocation-cost contrasts, take

significantly more turns in eight of the twelve contrasts, and significantly more local time in four. The mean Claude run costs 2.32 USD there against 1.06 USD on Hopsworks, and among runs with observed efficiency metadata the mean run consumes 67 turns and 667 seconds of local compute time against 30 turns and 304 seconds. Invalid runs concentrate in the Databricks runs as well, 35 of 262 against 16 of 264. The Databricks configuration therefore uses roughly twice the measured turns, model invocation cost, and local time, with no statistically detectable accuracy difference for any model except Mistral Large. Because platform, execution period, and coding agent version move together, the contrast does not identify a platform effect, and origin, vendor, architecture, and deployment model add further inseparable differences.

The model contrasts in Figure 4.9 identify one pronounced model-stack outlier rather than a general ordering among model tiers. Because the Mistral models carry no skills-condition cells, pairs at that condition exist only between the Claude models, leaving 28 pairwise model-tier contrasts within platform, interface, and skill condition. Of these, five are significant on the pass fraction after factor-wide Holm adjustment and five under McNemar's test on the success indicator, and every significant accuracy contrast involves Mistral Large on the losing side, while no contrast separates Claude Opus, Claude Sonnet, and Mistral Medium from each other on accuracy. The efficiency separation is broader, with 16 of 28 turn contrasts, 11 of 28 time contrasts, and four of the eight Claude cost contrasts significant. Three of the cost contrasts favor Claude Sonnet over Claude Opus on Hopsworks, where its runs cost half as much, while the fourth favors Claude Opus on the Databricks CLI, where Sonnet's far higher turn count reverses its price advantage. The supportable conclusion is therefore that the evaluated combination of Mistral Large and its agent produces the largest and most consistent accuracy separation on this benchmark, not that model tier formally outranks every other factor.

Figure 4.9: The model contrasts, pairwise between models within each platform, interface, and skills condition, decomposed as in Figure 4.6. Pairs at the skills condition exist only between the Claude models because the skills treatment was delivered to their agent only. Positive values mean the model named second has the higher value, and rows marked `+sk` are the skills condition. Every significant accuracy contrast involves Mistral Large.

The breakdown across the four machine learning operations (MLOps) task categories and the capstone projects in Figures 4.10 to 4.14 shows where the remaining differences sit, and each figure exposes a distinct pattern.

In the feature pipeline, shown in Figure 4.10, Claude Opus, Claude Sonnet,

and Mistral Medium sit at or near the 100% ceiling on both platforms, while Mistral Large is the outlier, dropping to 63% on the Hopsworks CLI and further to 19% on the Databricks CLI and 10% on the Databricks SDK.

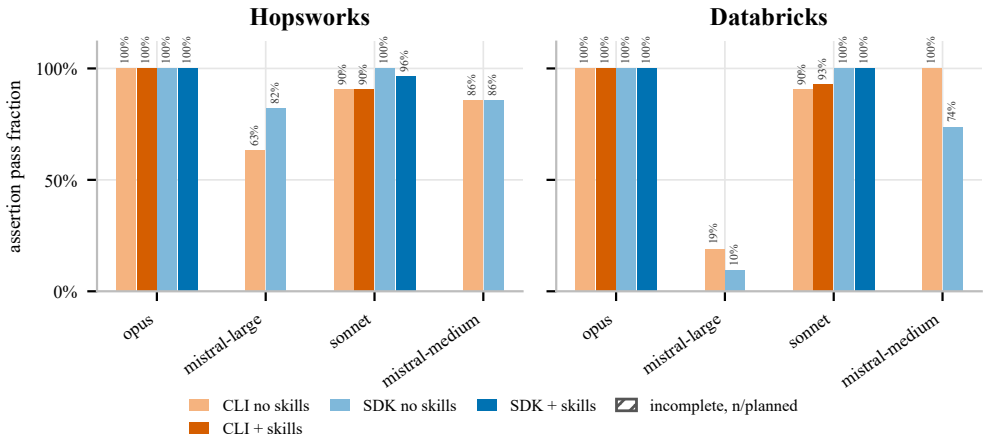

Figure 4.10: Mean assertion pass fraction of the feature pipeline tasks per model, interface, and skills condition. Invalid runs are scored as zero. The skills condition was delivered to the Claude agent only, so the Mistral models appear without it. Hatched bars are incomplete and report their retained and planned sample sizes. A cross at the baseline marks an incomplete bar whose observed value is zero.

Training tasks, reported in Figure 4.11, sit at 100% for every Claude cell on both platforms and for Mistral Medium on Hopsworks, and Mistral Large again collapses on the Databricks SDK to 17%, so training discriminates almost exclusively between Mistral Large on Databricks and everything else.

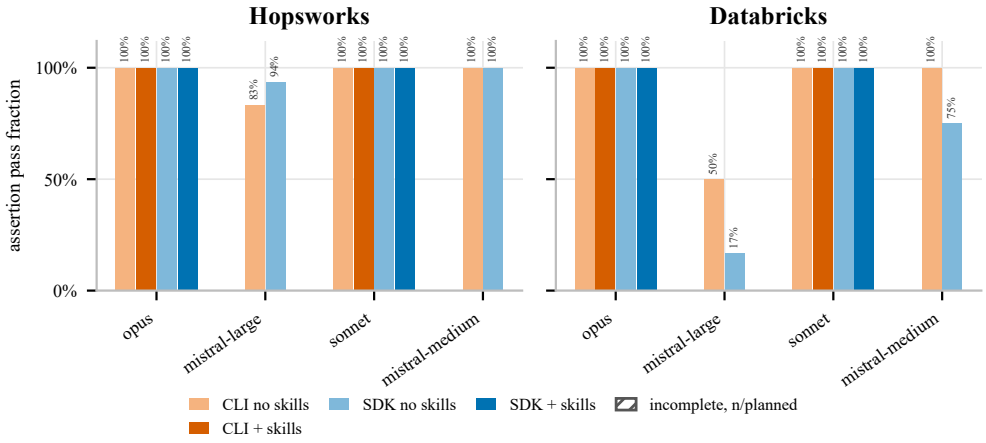

Figure 4.11: Mean assertion pass fraction of the training pipeline tasks, decomposed as in Figure 4.10.

The inference pipeline, plotted in Figure 4.12, shows the largest interface signal in the category breakdown, with the Hopsworks SDK passing 98% of these tasks against 77% for the CLI, consistent with inference rewarding the programmatic composition of deployment, request, and read back that the SDK expresses naturally.

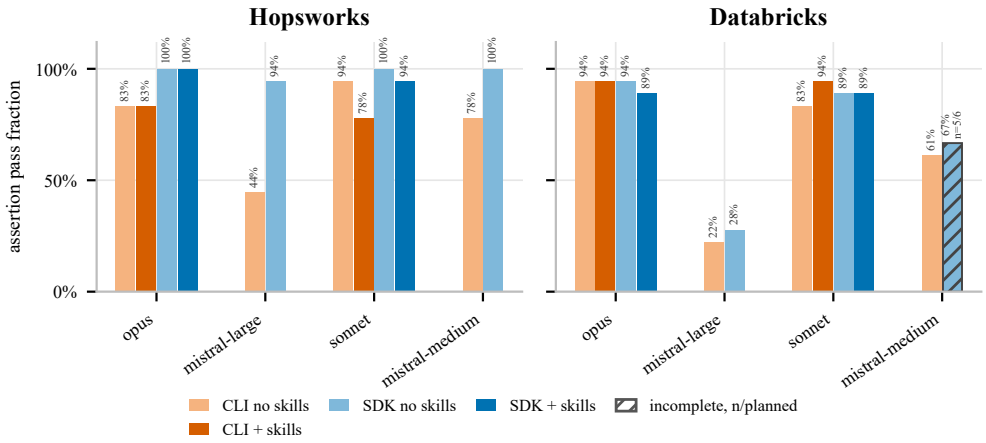

Figure 4.12: Mean assertion pass fraction of the inference pipeline tasks, decomposed as in Figure 4.10.

Operations tasks, displayed in Figure 4.13, are at 100% for every combination except Mistral Large, which falls to 67% on the Hopsworks CLI and to 33% on both interfaces of Databricks, so this category discriminates on the model axis rather than by interface or skills.

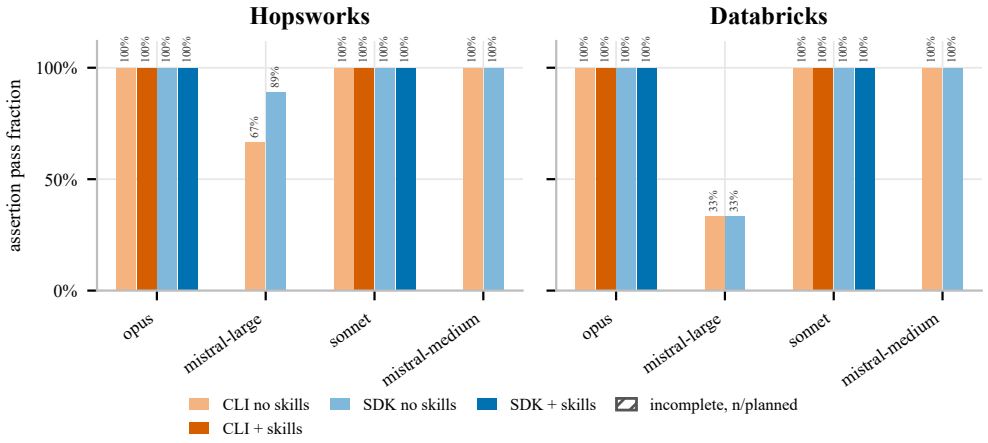

Figure 4.13: Mean assertion pass fraction of the operations tasks, decomposed as in Figure 4.10.

The two capstone projects, presented in Figure 4.14, which compose a full pipeline from feature engineering to serving, are the hardest task group on both platforms, with a mean pass fraction of 76% against 87% over the four MLOps categories. Claude Opus still passes 100% and Claude Sonnet at least 92%, whereas both Mistral models drop to 0% on the Databricks SDK and to at most 42% on the Hopsworks CLI. Competence on individual pipelines therefore does not fully compose into competence on their combination for the weaker models.

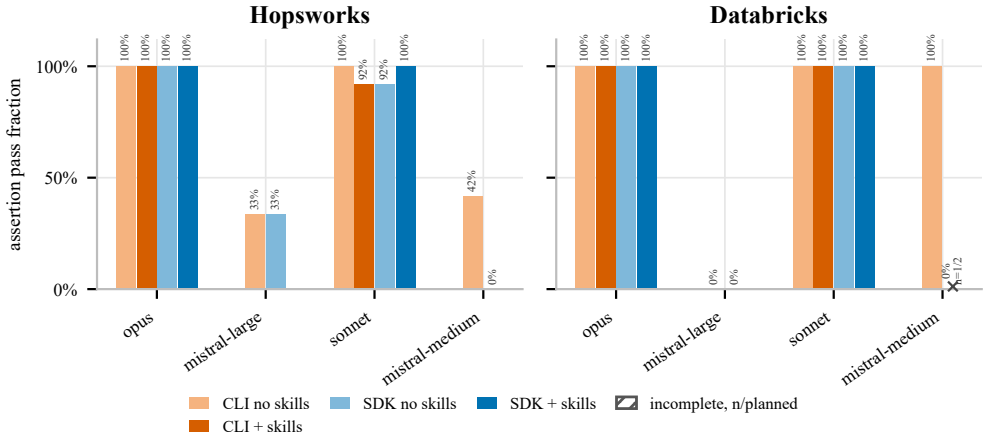

Figure 4.14: Mean assertion pass fraction of the capstone projects, decomposed as in Figure 4.10. The capstones are the hardest task group on both platforms.

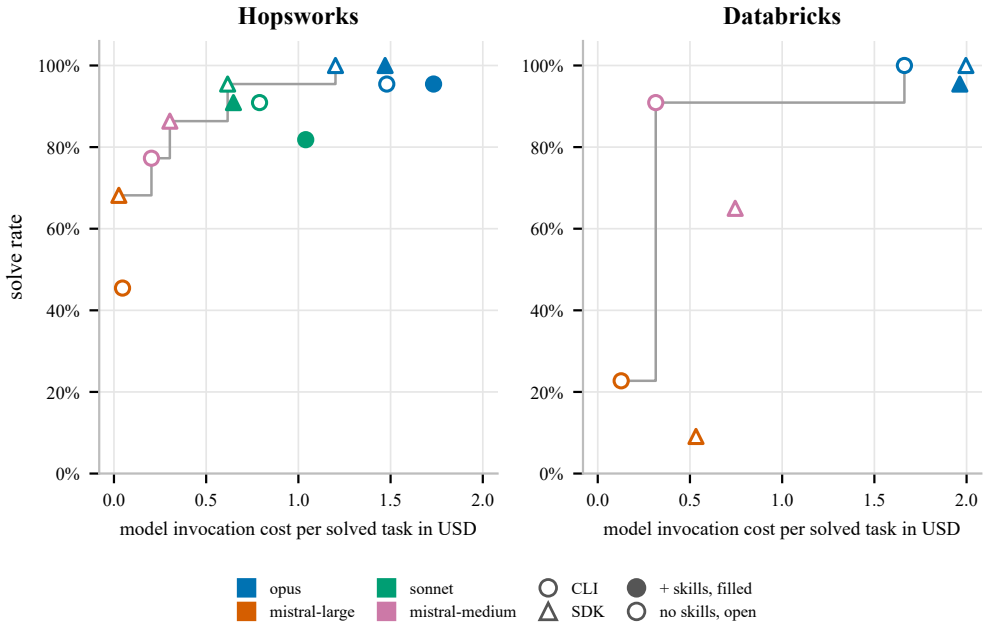

Figure 4.15: Solve rate against displayed model invocation cost per solved task on a linear scale for the low-cost range (0 to 2 USD), one marker per model, interface, and skills condition. Circles are CLI and triangles SDK cells. Filled Claude markers carry skills, and the Mistral models appear only in the none condition because the skills were not delivered to their agent. Claude costs are cache-inclusive, while Mistral costs are lower bounds because their cache usage is not retained. Platform infrastructure charges are outside this measure. The staircase is computed from the displayed values. Only comparisons among Claude markers identify cost dominance, and any segment determined by a Mistral lower bound is not an identified Pareto frontier.

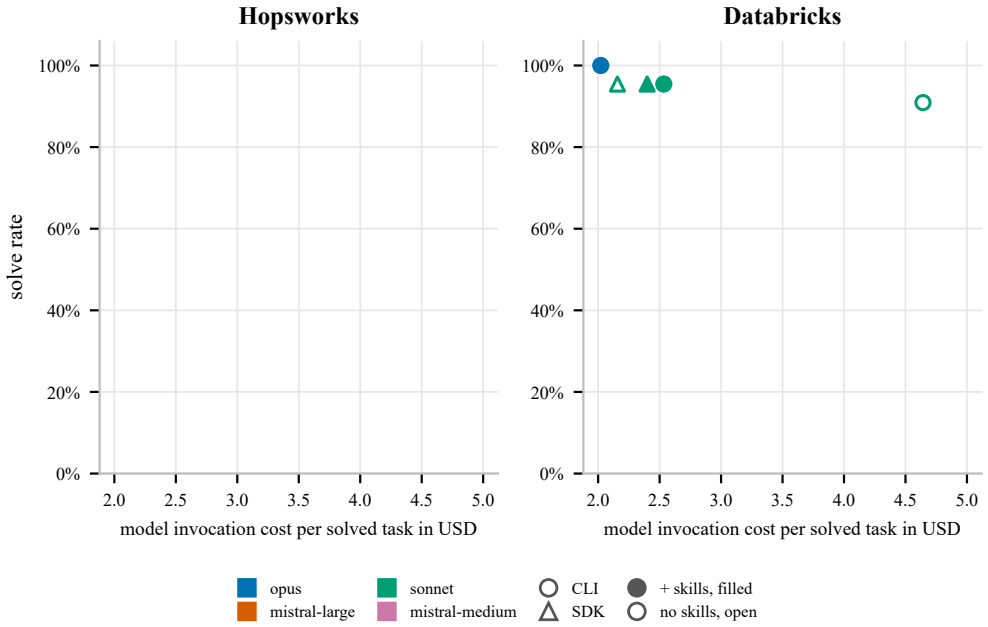

Figure 4.16: Solve rate against displayed model invocation cost per solved task on a linear scale for the high-cost range (2 USD and above), encoded as in Figure 4.15.

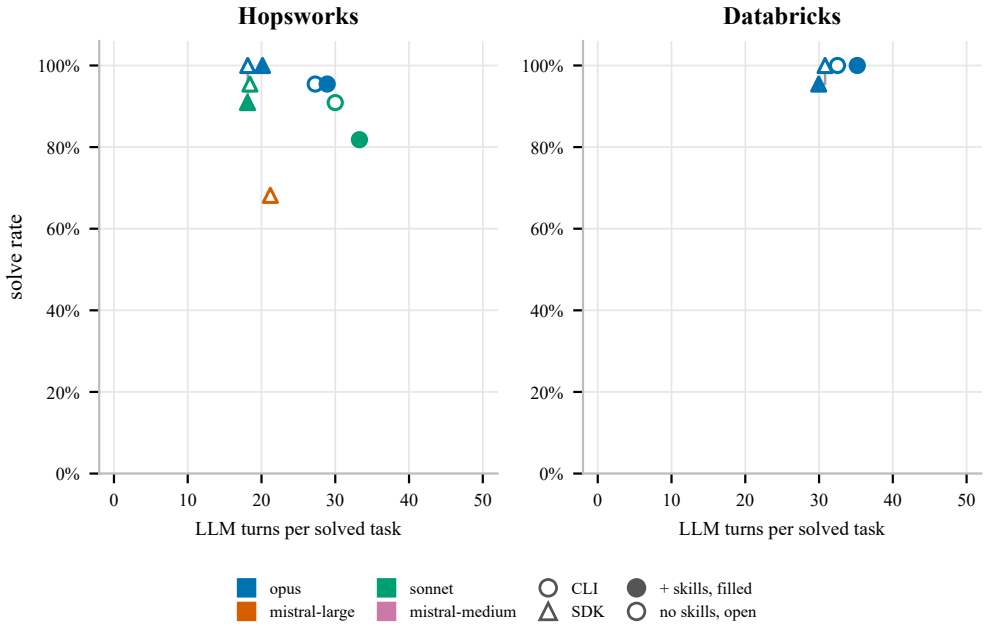

Figure 4.17: Solve rate against large language model (LLM) turns per solved task on a linear scale for the low-turn range (0 to 50 turns), encoded as in Figure 4.15.

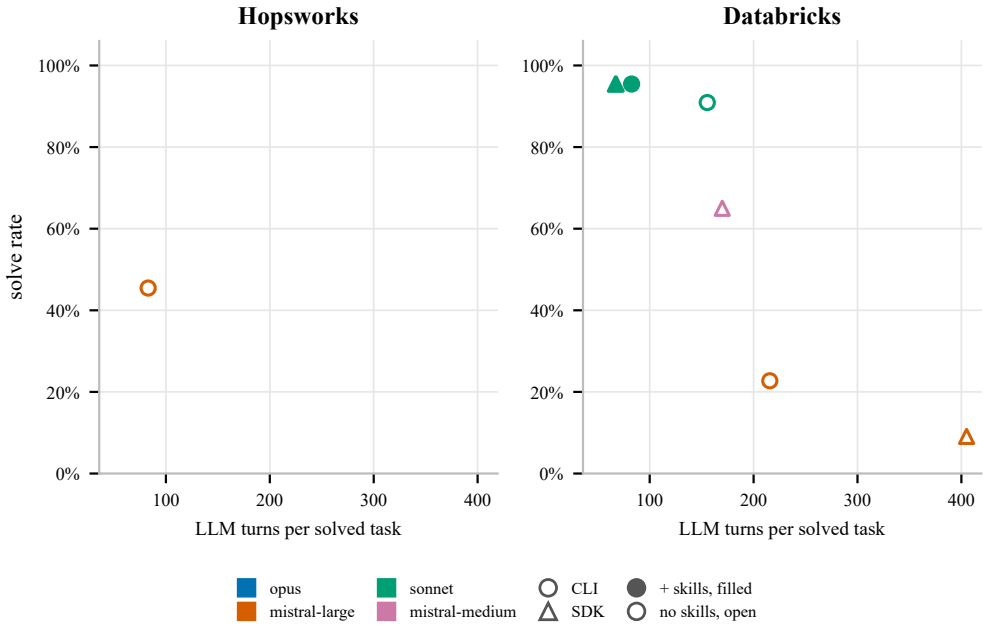

Figure 4.18: Solve rate against LLM turns per solved task on a linear scale for the high-turn range (50 turns and above), encoded as in Figure 4.15.

Figures 4.15, 4.16, 4.17, and 4.18 relate these outcomes to resource use. Under the cache-inclusive accounting, model invocation cost per solved task spans 0.62 to 4.64 USD across the Claude cells and is descriptively lower on Hopsworks than on Databricks. The Mistral points show only minimum possible costs because their cache usage is not retained, so they support neither a reliable cross-vendor cost ordering nor membership in a common cost frontier. Among the reliable Claude cells, Claude Opus buys near complete coverage at 1.2 to 2.0 USD per solved task. Deadline-killed runs lack final turn metadata, so turn-frontier points are shown only for cells with complete observations. Among those cells Claude Opus uses the fewest turns, solving Hopsworks tasks in about 24 LLM turns, and observed counts are descriptively higher on Databricks.

In answer to RQ2, the largest observed differences in what an agent accomplishes on a machine learning (ML) platform were associated with the combination of model and agent, followed by the platform configuration, and not with the interface or with the skills where they were delivered. The interface changes how the work is done rather than whether it succeeds. Two of the twelve turn contrasts significantly favor the SDK, no accuracy difference between the interfaces is statistically detectable in any contrast, and under cache-inclusive accounting the CLI holds no cost advantage, so none of the accuracy null hypotheses was rejected, two turn null hypotheses were rejected, and the directional cost expectation of the hypotheses is not supported. Skills, evaluated on the Claude agent, show no statistically detectable difference in accuracy in the eight Claude contrasts and produce only one significant efficiency contrast, fewer turns for Claude Sonnet on the Databricks CLI, so the accuracy null hypothesis was not rejected for the delivered skills contrasts and the data support no skills conclusion for the Mistral models. The earlier Hopsworks runs show no statistically detectable accuracy difference from the later Databricks runs for three of four models, exceed them for the fourth, and use roughly half the observed turns, Claude model invocation cost, and local time. This contrast does not isolate platform or platform origin because coding agent version and execution period move with the platform, while vendor, architecture, and deployment model also differ. The strongest capability limit of the benchmark appears where tasks compose into systems rather than in any single factor, since even the best cells lose more assertions on the capstones than on any single pipeline group.

4.3 Optimizations

The evaluation phase addresses RQ3 with thirteen further treatment grids, all on Hopsworks with Claude Opus, the platform and model pairing on which the comparison phase baseline is strongest. Because the optimizations are derived from and evaluated on the same task templates with a single model and platform, this phase is an exploratory case study of feasibility rather than a general test of agent driven interface optimization. The optimization phase produced six agent optimized command line interface (CLI) variants and one optimized skill set, each authored by Claude Code running Claude Opus from the run artifacts of the comparison phase. Table 4.1 lists the proposed changes. Each variant isolates one optimization on top of the pinned baseline CLI, and the optimized skills replace the original skills with condensed CLI first guidance. The evaluation executed every variant without skills, the optimized skills on the unmodified CLI, and every variant combined with the optimized skills extended by one further skill that documents that variant's optimization to the agent, each over the 22 benchmark tasks, and compares them against the treatment 1 baselines. Figure 4.19 reports accuracy, Figures 4.20 to 4.22 the efficiency metrics, and Figures 4.23 and 4.24 the planned optimization contrasts.

Table 4.1: The six agent optimized CLI variants and the optimized skills produced by the optimization phase. Each CLI variant isolates one change on top of the pinned baseline CLI.

Optimization	Proposed change
`opt1-batch`	Adds a batch command that executes many commands in a single process with a single login.
`opt2-session-reuse`	Resolves feature groups against the project's own feature store first and persists a cache of project and feature store identifiers across invocations.
`opt3-compact-json`	Emits compact JavaScript Object Notation (JSON) and switches to JSON output by default when standard output is not a terminal.
`opt4-idempotent`	Makes feature group creation idempotent through an if exists flag and rewrites errors to name the corrective command.
`opt5-quiet`	Adds a global quiet flag and strips null valued keys from JSON output.

Table 4.1: The proposed interface optimizations, continued.

Optimization	Proposed change
`opt6-stable-output`	Fixes the JSON key order and sorts listings deterministically.
optimized skills for each CLI optimization	Replaces the original skills with a condensed CLI first set of general platform guidance, written without task names or assertion specific steps to avoid overfitting the benchmark.

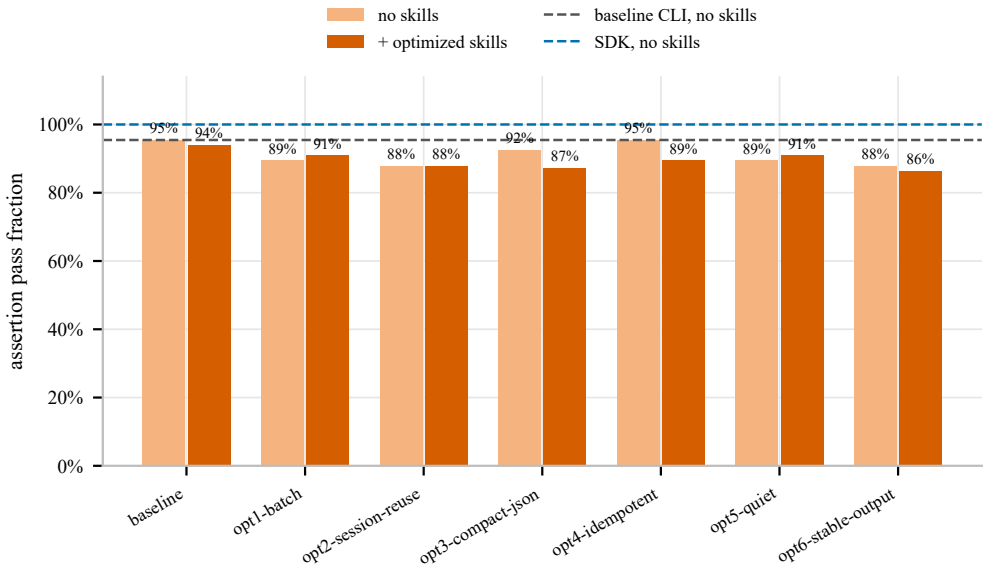

Figure 4.19: Mean assertion pass fraction of the optimizations on Hopsworks with Claude Opus, without skills and with the optimized skills. Invalid runs are scored as zero. The dashed lines mark the unmodified baseline CLI without skills and the software development kit (SDK) without skills.

No optimization exceeds the baseline in accuracy. The unmodified CLI passes 95% of assertions without skills, `opt4-idempotent` matches this value exactly, and every other optimization lies between 86% and 92%, with the combinations of variant and optimized skills occupying the lower half of that range. The optimized skills on the unmodified CLI reach 94%. The SDK reference sits at 100%, so no optimization closes the interface gap from above either. The efficiency picture in Figures 4.20 to 4.22 is equally flat. The baseline completes a task in a mean of 26 turns for 1.41 United States Dollar (USD) of cache-inclusive cost in 219 seconds of local compute

time, the variants without skills span 24 to 28 turns, 1.36 to 1.56 USD, and 211 to 278 seconds, and the combinations with skills reach 31 turns, 1.81 USD, and 326 seconds for `opt1-batch` with skills.

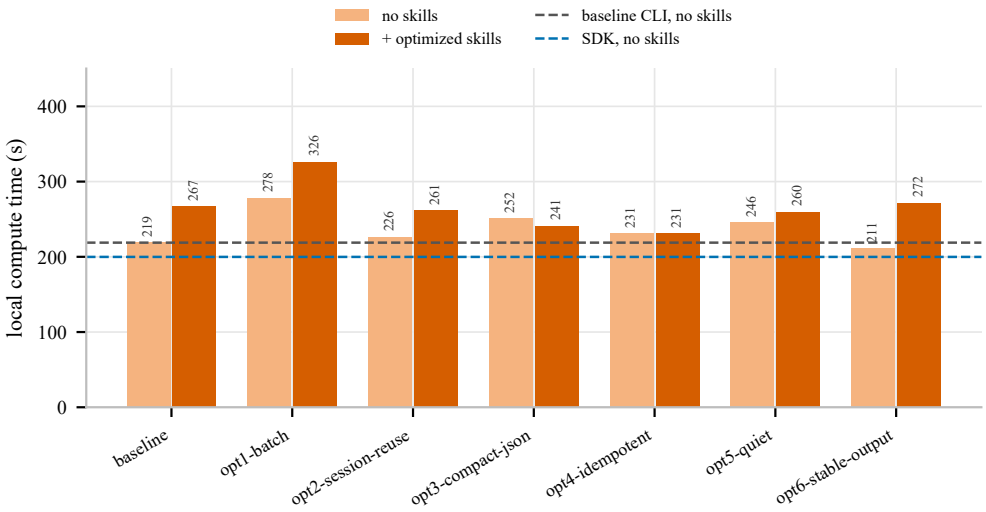

Figure 4.20: Mean local compute time per-task of the optimizations, decomposed as in Figure 4.19. The dashed lines mark the unmodified baseline CLI without skills and the SDK without skills.

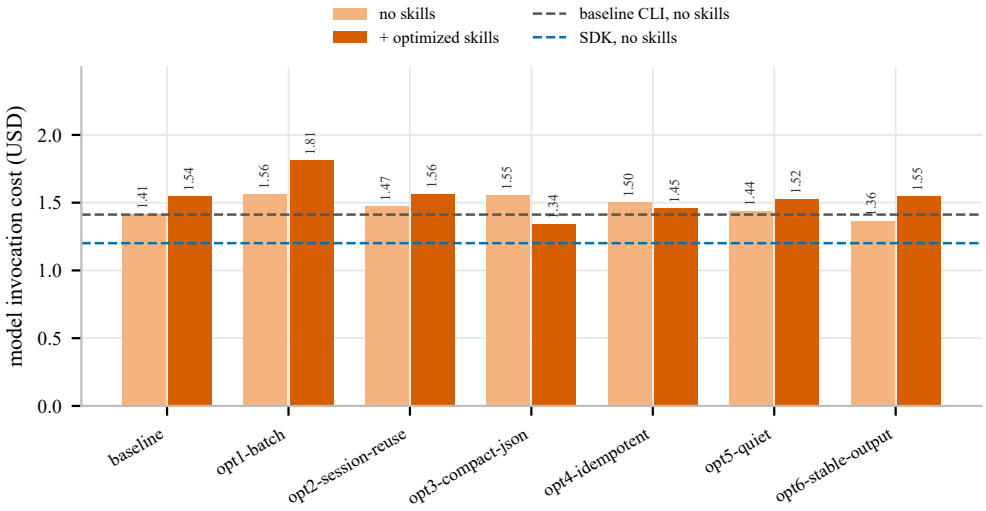

Figure 4.21: Mean model invocation cost per-task of the optimizations, decomposed as in Figure 4.20. Platform infrastructure charges are outside this measure.

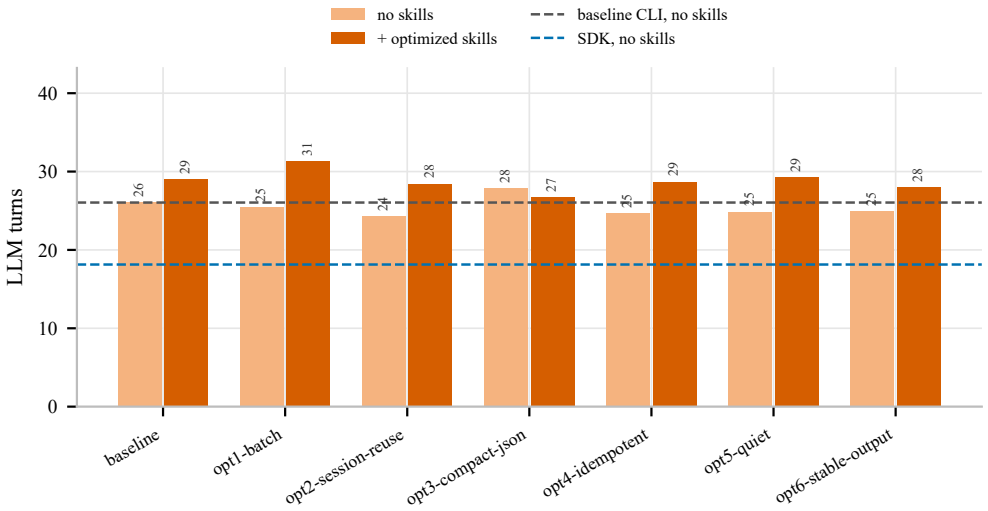

Figure 4.22: Mean large language model (LLM) turns per-task of the optimizations, decomposed as in Figure 4.20.

No statistically detectable difference was found between these treatments and their like-for-like CLI baselines. The bare variants and the optimized skills are tested against the unmodified CLI without skills, each variant with skills against the unmodified CLI with optimized skills, and the optimized skills additionally against the skills baseline promised in the method. The optimized set of skills reaches 94% assertion accuracy against 95% for both the skills and no skills baselines, and no CLI-baseline contrast is significant on pass fraction, success, model invocation cost, turns, or time after the factor-wide Holm adjustment. Against the SDK without skills, no optimization differs significantly in accuracy, cost, or time, while two combinations, the idempotence and quiet-output variants with optimized skills, take significantly more turns. The SDK descriptively uses fewer turns, but the factor-wide correction supports that difference for only those two combinations.

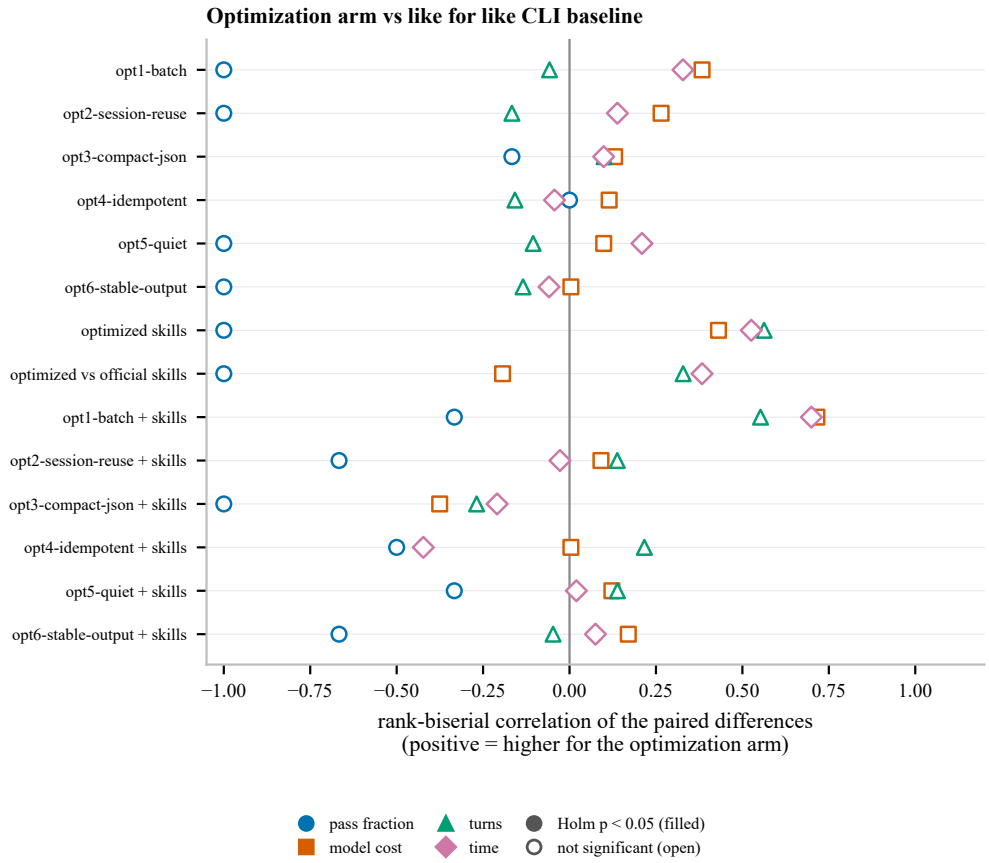

Figure 4.23: The optimization contrasts against like-for-like CLI baselines, each row a paired Wilcoxon contrast with rank biserial effect sizes and significance after Holm adjustment across all outcomes and both reference baselines. The bare variants and the optimized skills are compared against the unmodified CLI without skills, the optimized skills are additionally compared with the skills baseline, and each variant with skills is compared against the unmodified CLI with the optimized skills, isolating the CLI change at a fixed skill condition.

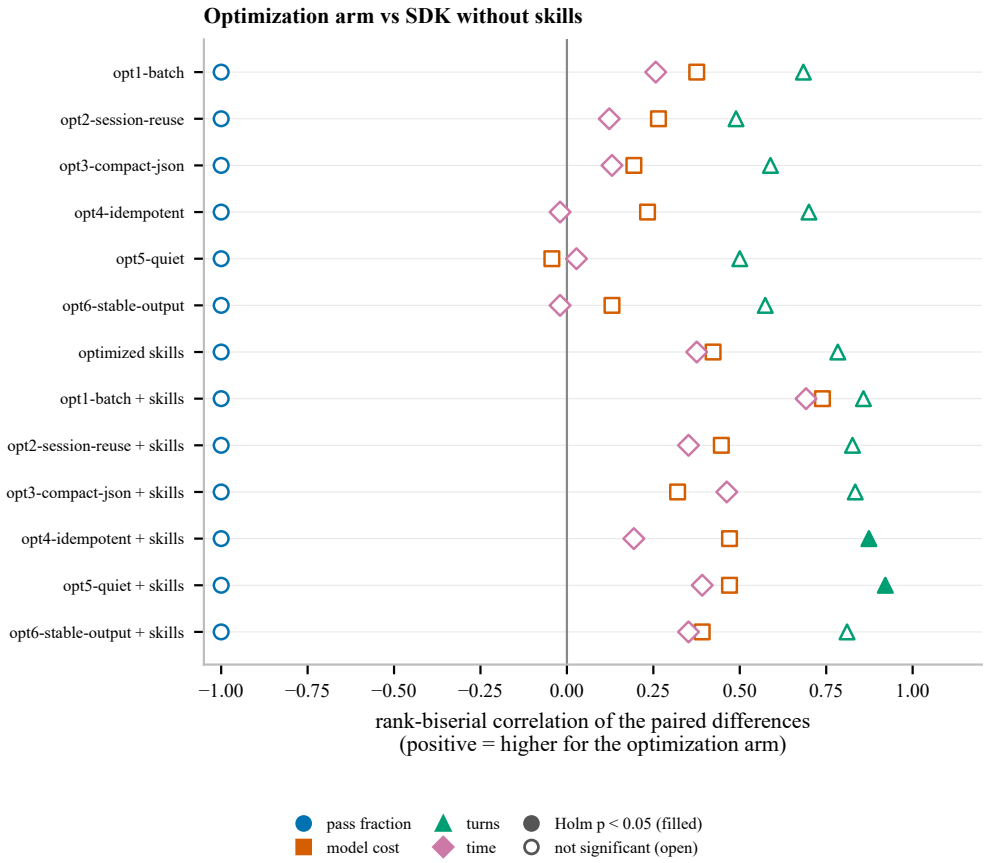

Figure 4.24: The optimizations against the SDK without skills, decomposed as in Figure 4.23, with significance using the same factor-wide correction, answering whether any optimization closes the interface gap.

In answer to RQ3, no statistically detectable difference was found between any optimization and its like-for-like baseline on any evaluated metric, including optimized skills against the skills baseline. None of the planned optimization null hypotheses was rejected. No optimization demonstrates an advantage over the native SDK, and two of the thirteen instead require significantly more turns than it after factor-wide correction. The baseline pairing of Claude Opus and the unmodified Hopsworks CLI already passes 95% of assertions in 26 turns per-task, leaving the optimizations little room to demonstrate value on this benchmark, and the observation that the added machinery of skills and batch execution makes runs descriptively heavier without making them better is itself the finding.

The next chapter discusses these results, reconstructs from the run artifacts why models underperform, and draws the practical implications.

Chapter 5

Discussion

This chapter interprets the results of the three research questions (RQs). It first discusses the benchmark results of RQ2, then reconstructs from the run artifacts why models underperformed where they did, and closes with the optimization results of RQ3 and their practical implication.

5.1 Benchmark Results

The clearest message of the benchmark is that the evaluated combination of Mistral Large and its agent produces the largest accuracy separation, while no statistically detectable difference in accuracy was found for the interface or for skills in the Claude cells where they were delivered. Differences also appear between the two platform configuration, but platform, execution period, and coding agent version are inseparable in that contrast, so it cannot identify an isolated platform effect. The practical result is therefore narrower than a general factor hierarchy. This benchmark exposes a pronounced weakness in one evaluated stack, while the contribution of platform architecture requires a frozen-harness comparison.

The model-tier comparison also contains a cost observation, though a chastened one after the accounting correction. Claude Sonnet passes 96% of assertions against 98% for Claude Opus, and whether its lower token price reaches the bill depends on the platform. On Hopsworks Sonnet's runs cost half as much as Opus's at similar turn counts, and three of the four cost contrasts there are significant in its favor, while on Databricks Sonnet consumes nearly three times the turns of Opus, and the cache traffic of those turns makes its runs costlier, significantly so on the command line interface (CLI). Among the Claude models, a lower per token price

therefore lowers the cost per completed run only while the interaction economy holds. Mistral's unrecorded cache usage prevents any reliable cross-vendor cost ordering, so its displayed lower bounds do not identify a cheaper alternative. Tier performance is not monotone within a vendor, as Mistral Medium outperforms Mistral Large by 36 percentage points on this benchmark, a gap driven by the larger model's grounding failures rather than by raw capability. Together these observations suggest that the model invocation price of agent operated platform work is governed as much by interaction economy, the turns and the cache traffic they imply, as by per token rates, which matters directly for the resource constrained deployments that sovereign artificial intelligence (AI) strategies emphasize.

The interface result deserves a mechanical reading. The software development kit (SDK) descriptively uses fewer turns, and two of twelve turn contrasts are significant. No accuracy difference between the interfaces is statistically detectable anywhere, and under cache-inclusive accounting the CLI holds no cost advantage either. The run artifacts show why. A CLI agent works in many small steps, issuing one command per turn, reading its short output, and deciding the next command, while an SDK agent batches whole phases of a task into a single script whose execution consumes one turn. The directional hypothesis expected the progressive disclosure of the CLI to save both cost and turns, and the corrected data support neither saving. Every turn re-reads the growing cached context at the cache read rate, so the many small steps of the CLI multiply exactly the traffic that cache billing prices, and the initial appearance of a CLI cost advantage was an artifact of accounting that omitted this traffic. The turn economy favors the SDK, while the model invocation cost data provide no evidence for the expected CLI saving. The conditional form of this finding is the practical one. Where a platform's CLI and SDK are similarly mature and coverage matched, as they are here after the parity correction, progressive disclosure yields no statistically detectable benefit in accuracy or cost, the CLI did not reduce turns, and a cheaper interface has to be demonstrated rather than assumed. The accuracy null itself carries the ceiling caveat, since Claude Opus already passes 98% of assertions on either interface, so a harder benchmark that separates the strongest models could still uncover an interface accuracy signal that this one cannot resolve.

The skills result also has a mechanical explanation in the artifacts, and a cautionary one. For the Claude models the skills are largely redundant on this benchmark, since Claude Opus passes 98% of assertions without them, leaving no headroom for guidance to add value, and the only efficiency contrast surviving factor-wide correction is a turn reduction for Claude Sonnet on the

Databricks CLI. A harder benchmark that pushes the strong models off the ceiling could restore this headroom and might expose an accuracy contribution of skills that this benchmark cannot detect. For the Mistral models the audit found the skills were not ignored but absent, since the meta-harness installed the skills where only the Claude agent discovers them, so what the design intended as a skills comparison across both agent vendors was in fact a Claude only one. An earlier reading of the Mistral logs, that the weaker models fail to consult deployed guidance, does not survive this finding, since guidance that never enters the agent's context cannot be consulted. What remains supportable is narrower. Written guidance did not lift models that already verify their work, and whether it lifts the models that would need it most is exactly the question the Claude-only evaluation leaves unmeasured.

The platform contrasts show an interaction-economy difference between the two platform configurations. For three of the four models no accuracy difference is statistically detectable, while the Databricks runs consume roughly twice the turns, measured Claude model invocation cost, and local time of the Hopsworks runs and also produce roughly twice the invalid runs. The artifacts provide a plausible platform-side mechanism. Agents on Databricks spend turns provisioning warehouses, waiting on statement execution, and navigating the catalog and schema hierarchy before task logic runs, work that has no counterpart on the smaller platform. Because the coding agent version moved with the platform, however, the artifacts do not make the aggregate difference an identified platform effect. The fourth model turns this difference into a capability cliff, examined next.

5.2 Model Underperformance

The collapse of Mistral Large on the Databricks software development kit (SDK), 18% against 84% on Hopsworks, was initially suspected to have a dependency cause, namely that feature store work on Databricks requires a second package beyond the general SDK, the dedicated feature engineering client, and that Mistral failed to discover it while the Claude models did. Inspection of the retained benchmark-task artifacts rejects this explanation. The feature engineering package appears only in a small minority of the Claude Opus SDK runs, concentrated in the capstones, while Opus passes the remaining tasks with the general SDK and plain structured query language (SQL) statement execution. The differentiator was not which package the agent found but how it worked. Opus begins by enumerating the real application programming interface (API) surface

of the client object, writes a polling helper so that every SQL statement is driven to completion and checked, and reads its own deliverable back before declaring success. Mistral Large instead repeatedly hallucinates method signatures, writes to the deprecated workspace file system root that the platform has disabled, fires statements without checking their state, and ends failed runs by asserting that the deliverable exists when it does not. One batch scoring run even performed an explicitly mocked table write and then reported the task complete. After several such errors the model tends to conclude, falsely, that the SDK does not support the operation, and downgrades to a local file or a prose recommendation.

The same behavior explains the command line interface (CLI) weakness of Mistral Large on both platforms. Its logs show hallucinated legacy command syntax from a predecessor version of the Databricks CLI, and a characteristic quoting failure in which backtick quoted SQL identifiers inside double quoted shell arguments open command substitution, so the sandbox's interface allowlist rejects fragments of the SQL text as foreign commands. Rather than switching quoting strategy, the model escalates to denied tools and then submits a missing capability report, in one run deleting the feature group it had already created before giving up. The near parity of Mistral Large on the Hopsworks SDK, where it writes plain Python against a single client, indicates that the observed deficit is associated with grounding and verification discipline rather than the benchmark's machine learning (ML) content, and it suggests that scores earned through an SDK do not transfer automatically to CLI operation for weaker models.

The remaining underperformance has three smaller anatomies. First, every one of the eleven budget timeouts in the committed grids belongs to a Mistral model, seven of them on the two capstones, and the artifacts show Mistral Medium being killed by the one hour budget with the feature group, training dataset, and registered model already on the platform but the predictions not yet written, so the capstone deficit of the weaker models is partly time pressure on composition rather than inability. Second, Mistral Medium's weakest cell, the Databricks SDK, is not the discovery failure of its larger sibling but verification negligence, since it finds the correct APIs quickly, then leaves tables empty after failed inserts and reports success after checking only that the table exists, never its row count. Third, the isolated 33% cells of Claude Sonnet on Hopsworks dissolve under inspection into one run killed by a mid response connection failure and one genuine off by one in a ranking query, a reminder that with one run per cell the extreme cells of the grid carry sampling noise, which is why the analysis is based

on paired contrasts over 22 tasks rather than on individual cells.

The capstone projects deserve their own note, since they are the only structural weakness shared by all models. Even cells that pass every single pipeline task lose assertions on the capstones, and the failing assertions are the late ones, the prediction table, the metric threshold, and the registry entry, never the early feature engineering. Composing pipelines into a system multiplies the surface on which one unverified step silently invalidates the rest, which is precisely the failure mode that the read back grading of this benchmark is designed to expose.

5.3 Optimization Results

The optimization result of RQ3 is a null with a sharp practical edge. All six command line interface (CLI) variants encode changes that practitioner intuition and the agent's own optimization phase considered beneficial, batching, session reuse, compact output, idempotence, quietness, and determinism, and the optimized skills condensed platform guidance into the form the literature recommends. None produced a statistically detectable improvement on any evaluated metric against its like-for-like baseline. Part of the explanation is a ceiling effect, since the baseline pairing of Claude Opus and the unmodified CLI already passes 95% of assertions in 26 turns, and part is that the optimizations address frictions a strong model already routes around, while the model whose failure modes they would target, Mistral Large, was never part of the evaluation phase.

The practical edge is a warning about unvalidated optimization. Every optimization here looked reasonable before measurement, and several practitioners would ship any of them as an obvious improvement, yet none differed detectably from its paired baseline. In most production settings no benchmark exists to catch this, since teams adopt agent-facing interface changes, prompt additions, and skill files on plausibility alone, and the cost of a counterproductive change is diffuse, a few percent more tokens and turns on every task, invisible without paired measurement. The seeded, read back graded design of MLPlatformAgentBench (MLPAB) is deliberately cheap to rerun, and this experiment demonstrates the workflow it enables, treating every agent-facing change as a hypothesis that must beat a paired baseline before it ships. The absence of such regression measurement for agent interfaces is, on this evidence, not a hypothetical risk but the default state of practice.

The final chapter concludes from these findings, states the limitations of the work, and outlines future work.

Chapter 6

Conclusions and Future Work

This chapter closes the thesis. It draws the conclusions from the results of the three research questions (RQs), states the limitations that qualify them, outlines the future work that follows from both, and reflects on the economic, social, environmental, and ethical aspects of the work.

6.1 Conclusions

This thesis asked which machine learning (ML) platforms a coding agent can operate today, how the interface and its surrounding artifacts shape that operation, and whether an agent can optimize its own operating interface. It answered these questions by constructing MLPlatformAgentBench (MLPAB), a benchmark of 22 seeded task templates across the feature, training, and inference pipelines, operations, and two capstones, each graded by reading the produced state back through the platform's own data paths, and by executing the committed treatments, eight comparison grids across two platforms, four models, two interfaces, and two skill conditions, and thirteen evaluation grids for the optimization study.

For **RQ1**, the assessment of 39 candidate platforms showed that agent-operable coverage of machine learning operations (MLOps) and automated machine learning (AutoML) tasks is concentrated in five leading platforms dominated by American vendors. The software development kit (SDK) remains the most complete interface, while the Model Context Protocol (MCP), despite being agent-specific, reaches full stage coverage on no platform. ML platform agents, defined as vendor-built agents that carry out the feature, training, and inference workloads of their own platform, exist on four platforms and reach full stage coverage on two, while skills exist for

only eight vendors and a documented meta-harness on three, Omnigent on Databricks, the Hops Terminal on Hopsworks, and CoCo on Snowflake [248]. Full coverage through both a command line interface (CLI) and an SDK, the precondition for the experimental interface comparison, exists on exactly three platforms and on exactly one platform of European origin. The assessed landscape an agent can operate is therefore real but narrow, and its supporting ecosystem of deployment infrastructure and skills is younger still.

For **RQ2**, the largest observed differences in what an agent accomplishes are associated with the combination of model and agent, with no statistically detectable difference in accuracy for the interface or the delivered skills. Claude Opus solves 98% of its runs and Mistral Large 36%, a gap wider than any produced by the other measured conditions. The interface changes how work is done rather than whether it succeeds, since two of twelve turn contrasts significantly favor the SDK, no accuracy difference between the interfaces is statistically detectable in any contrast, and under cache-inclusive accounting the CLI holds no cost advantage. Where a platform's CLI and SDK are similarly mature and coverage matched, progressive disclosure therefore yields no statistically detectable benefit in accuracy or cost, and the CLI did not reduce turns. Skills, evaluated on the Claude agent, show no statistically detectable difference in accuracy within the eight Claude contrasts, plausibly because the Claude models already operate near the ceiling without them. The earlier Hopsworks runs show no statistically detectable accuracy disadvantage for three of four models, an advantage for the fourth, and use roughly half the measured turns, Claude model invocation cost, and local time of the later Databricks runs. Because coding agent version and execution period move with platform, this is not an isolated platform or origin effect. The model-stack gap is the larger observed sovereignty-relevant difference in this dataset, but origin remains confounded with vendor, model class, and agent harness.

For **RQ3**, an agent can produce plausible optimizations of its own operating interface, but on this benchmark no statistically detectable difference was found between any optimization and its like-for-like baseline on any evaluated metric, including optimized skills against the skills baseline. Feasibility is demonstrated in an exploratory in sample setting, benefit is not. The constructive reading, developed in the discussion, is that every agent-facing change should be treated as a hypothesis to be tested against a paired baseline, and the benchmark built for this thesis is precisely the instrument that makes such regression measurement cheap.

Beyond the answers to the research questions (RQs), the thesis leaves three durable artifacts. The market assessment matrix documents the agent-facing

surface of 39 platforms with per criterion sources. MLPAB is a meta-harness for benchmarking agents that operate and optimize ML platforms. It wraps each coding agent in a sandboxed, policy-enforced session behind a uniform application programming interface (API), adds seeded task generation, state based grading, and recorded per-run provenance, and extends to further platforms, models, and interfaces at the cost of an adapter. And the capability parity correction contributed upstream to the European platform's CLI remains available to every user of that interface.

6.2 Limitations

The validity analysis of the method chapter states the threats that were accepted at design time, and the results sharpen several of them into concrete limitations. Statistical power comes from pairing across the 22 tasks rather than from repetition, since every cell holds a single run. The inspection of the extreme cells in the discussion showed what this means in practice, as isolated cell values as low as 33% dissolved into one connection failure and one genuine off by one error, so individual cells of the grid must not be read as stable estimates, and only the paired contrasts carry evidential weight. Because the tasks are an authored fixed population, that evidence extends to this benchmark, and generalization beyond it is an argument from the benchmark's construction rather than from the tests. Two further qualifications apply to the analysis itself. Task instances are seeded per treatment, so only the interface and skills contrasts compare identical instances, while the model, platform, and optimization contrasts compare different seeded instances of the same templates, leaving instance difficulty in their error term. And while the test choices were fixed in the method, the analysis implementation postdates execution and no equivalence margins were defined, so null results throughout this thesis mean no statistically detectable difference rather than demonstrated equivalence. Zero differences discarded by the signed-rank test can reduce the effective sample, so the reports state both paired and non-zero counts. Deadline-killed runs retain zero accuracy, while their zero turn and time placeholders are excluded from efficiency analysis because final metadata was not emitted, reducing affected turn and time contrasts to 17 pairs. A third qualification concerns cost. A post execution audit found that the original cost accounting omitted the separately billed prompt cache usage, understating Claude costs by more than a factor of four and initially producing an apparent cost advantage of the command line interface (CLI) that did not survive correction. Claude costs were re-derived from the

retained transcripts, Mistral costs remain lower bounds, and cost conclusions are based on Claude contrasts only. Moreover, the measure covers model invocation charges rather than platform compute, storage, warehouse, cluster, or networking costs, so it is not total cost of ownership. The episode is itself an instance of the measurement discipline this thesis advocates. A fourth qualification concerns treatment delivery. The skills were installed at the Claude coding agent's discovery location and the preflight skill probe was skipped for the other agents, so the Mistral agent never received the skills condition, a failure found in the same audit. The skills conclusions are therefore Claude only, the cross-vendor model contrasts at the skills condition were excluded as not treatment equivalent, and verifying treatment delivery per coding agent is a lesson the meta-harness carries forward. A fifth qualification concerns the coding agent versions, which were not pinned and, for Mistral Vibe, not recorded. The Claude runs drifted from patch release 2.1.179 to 2.1.197 between the earlier Hopsworks and later Databricks executions, and the intervening changelog includes behavioral changes to the harness. The platform findings are consequently comparisons between the earlier Hopsworks runs and the later Databricks runs in which agent version and execution period move with platform, and the size of that confound cannot be estimated from these data. Pinning and recording the coding agent version joins the audit findings as a requirement for future runs.

The comparative conclusions inherit the confounds of the design. Models run inside their vendors' coding agents, so every model difference is a difference between combinations of model and agent, and the collapse of Mistral Large on the CLI, driven by hallucinated command syntax and quoting failures, may partially reflect its harness rather than the model alone. The one hour budget interacted with model speed in a way that the results made visible, since all eleven timeouts belong to Mistral models and seven of them cut capstone runs that had already built most of the pipeline, so the capstone deficit of the weaker models conflates ability with time. The committed grid covers two platforms and four models, and Open Artificial Intelligence (OpenAI) models were not executed, so the model-stack and platform conclusions are strongest for the eight committed grids and indicative beyond them. The Hopsworks interface contrast likewise compares the contribution-fork `5.1+alerting` CLI with the distinct upstream `5.0.0` software development kit (SDK), so it establishes coverage-matched interface configuration differences rather than an entry-point-only effect. The command sandbox layer adds two smaller environmental differences stated in the method, since it exists only in the Claude coding agent and, within the

Claude conditions, the Databricks CLI binary had to be excluded from it, so the interface conditions were not byte identical environments. Uniform container-based sandboxing, which was not possible on the local execution host, would remove both differences and is named in future work. Runs were deterministically ordered rather than randomized or counterbalanced, leaving condition order exposed to short-lived provider and platform changes.

The optimization results carry the narrowest scope. The variants were authored by and evaluated with a single model on a single platform's CLI, on the same task templates from which they were derived, and no tasks are held out. The evaluation showed a ceiling effect, since the baseline already passes 95% of assertions, and the discussion noted the resulting blind spot, namely that the optimizations most likely to matter are those targeting the failure modes of weaker models, such as a quoting robust command surface, which the evaluation phase never paired with a weaker model. The null result of RQ3 is therefore a statement about optimizing an interface that a strong model already operates well, not about agent driven interface optimization in general. The comparison also reuses the treatment 1 baseline executed with Claude Code 2.1.179–2.1.183 against optimizations executed later with version 2.1.185, so it is a chronological baseline-versus-variant configuration in which the interface artifact, task seed, execution period, and coding agent patch level are not fully separable.

A further limitation concerns the discrimination power of the benchmark against state of the art (SOTA) models. The category breakdown showed that Claude Opus, Claude Sonnet, and Mistral Medium sit at or near the 100% ceiling on the feature, training, and operations tasks on both platforms, so within those categories the benchmark cannot separate the strongest models from each other and is therefore too weak to discriminate the frontier of current approaches. The remaining discrimination comes from the inference tasks, where the interface gap is visible, and from the capstones, where composed pipelines still separate models. As frontier models continue to improve, more of the current tasks are likely to saturate, and preserving discriminative power will require harder templates, larger data volumes that force the distributed execution paths of the platforms, or adversarially authored task content, all named in the future work section.

Finally, the benchmark itself carries provenance caveats stated in the method chapter. The benchmark is organized around the feature, training and inference (FTI) architecture associated with Hopsworks, the capstone projects derive from that vendor's public example systems, and the tasks were generated with a Claude model whose model peers are among the evaluated

subjects. The mitigations, platform neutral prompts, per platform adapters, state based grading, computable ground truths, and a cross-vendor review, remove the mechanical forms of this proximity but not the distributional one. The platform contrast, the capstone results, and the sovereignty discussion are therefore descriptive of the evaluated pairings and do not identify a general effect of geographic origin.

6.3 Future Work

Future work divides along the three roles that MLPlatformAgentBench (MLPAB) plays in this thesis, as a benchmark, as a meta-harness for benchmarking agents that operate and optimize machine learning (ML) platforms, and as an instrument for interface optimization.

As a benchmark, the most pressing extension is to raise its difficulty ceiling. The category breakdown showed that Claude Opus, Claude Sonnet, and Mistral Medium sit at or near the 100% ceiling on the feature, training, and operations tasks on both platforms, so this benchmark cannot separate the strongest models from each other on those categories, and any future frontier model will likely saturate the remaining ones. A next-generation benchmark should be hard enough that even the highest performing model achieves only a low pass fraction, so that the metric retains discriminative power across model releases. Concrete levers include composing longer end-to-end pipelines that magnify late-stage failure, introducing scale that forces distributed execution, and authoring tasks whose intended solution deliberately falls outside the priors of current frontier models. The remaining extensions raise realism alongside difficulty. The generated datasets are small by design so that runs stay within budget, and scaling the task instances to terabytes would test whether agents can operate the distributed execution paths of the platforms rather than their convenience paths. The tasks could also carry more caveats of the kind that separate careful from careless engineering, though this is harder than it sounds, since caveats authored by a frontier model are plausibly within the priors of that model class, so genuinely adversarial task content has to come from practice. Given time and access, collecting real use cases from platform customers, ideally the erroneous and abandoned ones that agents and engineers alike have failed to crack, would replace authored difficulty with encountered difficulty. In the same direction, the model tier question deserves systematic treatment. Claude Sonnet delivers near frontier accuracy here, and its lower token price halves the cache-inclusive model invocation cost per run on Hopsworks

while its higher turn count makes it the costlier model on Databricks, so mapping how low a product tier can be while still operating a platform reliably remains necessary to identify the real price floor of agent operated ML work. A complete economic evaluation should additionally combine model invocation charges with platform compute, storage, warehouse, cluster, and networking charges, and report end-to-end wall-clock time alongside the local interface measure. Beyond single tasks, the benchmark could become multidimensional, including collaboration with real users and with other agents, resolving support tickets end-to-end, and observing how ML engineers actually interact with an autonomous colleague. A further step would simulate the day-to-day of an ML engineer over days and weeks, with multiple tools, synchronous and asynchronous workloads, long running training jobs, stakeholder communication, and the continuous improvement of deployed models. At the far end, as agents outgrow benchmarks that seemed unsolvable only a few years ago, an economic benchmark suggests itself, in which the agent takes customer requests, turns them into tasks, builds and serves a model, and is measured by whether the result sells.

As a meta-harness for benchmarking, the natural next comparison is between the two routes the market assessment scores separately, the vendor-built platform agent and the coding agent hosted in a meta-harness. A vendor agent's narrower scope may buy security and traceability advantages that this thesis could not measure, and its existence is motivated by exactly the difficulty this work confirmed, namely that constraining a general coding agent to one interface is hard.

The inverse experiment is equally interesting, running agents without the allowlist and measuring when and how they break out of the intended interface or search for workarounds, which turns the enforcement layer from a control into an object of study. A stronger form of this is real sandboxing, in which the environment rather than a tool list bounds the agent, which would also give every coding agent and interface an identical containerized environment, removing the command sandbox asymmetries that the local setup of this work accepted, and where it becomes testable whether prompting alone can hold a fully intelligent agent to its goals. Because nothing in the meta-harness is specific to ML platforms beyond the adapters, building equivalent benchmarks for other industries and platforms would show whether the findings on interfaces, skills, and model origin transfer, and the comparison of American and European coding agents could be repeated wherever such benchmarks exist. Within the present domain, the three remaining leading platforms, Amazon SageMaker, Microsoft Azure

Machine Learning, and the Google platform, are already supported by adapters and await full grids, newer model generations could be evaluated in full grids, and the Model Context Protocol (MCP) could be benchmarked in the individual pipelines where vendors do cover it, even though no vendor covers all of them. Finally, a controlled deployment in which Databricks and the other platforms run on the same hardware as Hopsworks, with the coding agent executable pinned across all treatments, would remove both the managed-service asymmetry and the version confound and allow the platform architectures themselves to be compared.

For optimization, the design of this thesis was a single shot across two variables, the command line interface (CLI) and the skills, and its null result motivates iteration rather than abandonment. One extension optimizes a single variable over multiple rounds, feeding each round's benchmark results back to the optimizing agent and testing whether the interface converges toward a measurable improvement. A second gives the agent the interface source code and a live platform and asks it to discover new capabilities rather than polish existing ones. The central difficulty of both is the one this thesis leaves sharpest, namely finding an evaluation target that the optimizing agent cannot overfit, since any fixed benchmark that guides optimization eventually stops measuring generality, and held out tasks, rotating seeds, and real workloads are the obvious but untested remedies.

6.4 Reflections

The work carries economic, social, environmental, and ethical aspects beyond its technical questions, and this section reflects on each in turn.

Economically, the thesis puts numbers on a shift that is already under way, the transfer of machine learning (ML) platform operation from specialist working hours to managed model invocations. The committed runs completed 22 realistic machine learning operations (MLOps) and automated machine learning (AutoML) tasks per grid at a model charge of roughly one to three United States Dollar (USD) per solved task for the capable models, a price at which routine platform work becomes accessible to organizations that cannot staff a platform team. The same accounting carries a warning. The initial cost figures understated the true bill by more than a factor of four because prompt cache traffic was not priced, so organizations that adopt agents on the strength of unaudited cost dashboards can misjudge their spending badly, and the cache-inclusive accounting developed here is directly reusable for that audit. The measured interaction economy

also shows that the cost of agent work is governed as much by turns and cache traffic as by per token prices, which makes cost a design property of interfaces and not only a procurement decision.

Socially, the sovereignty dimension of RQ2 addresses a question European organizations face under data residency and strategic autonomy constraints, namely whether choosing European technology sacrifices capability. The results give a differentiated answer, since the European platform matched the American one in accuracy for three of the four evaluated models, while the European models trailed the American ones as complete combinations of model and agent. Evidence of this kind supports procurement and policy decisions that would otherwise rest on vendor claims. The work also touches the changing role of the ML engineer, whose platform work the agents here perform end-to-end, shifting the human contribution toward specifying, reviewing, and measuring agent work. The thesis itself, whose use of large language models (LLMs) is disclosed in the acknowledgments, is a first hand instance of that shift.

Environmentally, LLM inference consumes energy, and the experiments consumed 175 hours of agent compute and 37.2 million fresh tokens plus a substantially larger volume of cache traffic. Three design choices limit and justify this footprint. The generated datasets are deliberately small, so no run trains at production scale, the seeded benchmark makes every further comparison cheap rather than requiring new exploratory runs, and the null result of the optimization study shows why such measurement matters at large, since unvalidated interface changes that add a few percent of tokens and turns to every task would waste energy invisibly at deployment scale, and paired regression measurement catches them before they ship.

Ethically, the central concern is an intelligent agent operating a production grade system. The experiments confine the agents through an allowlist and treat holding an agent to its intended interface as an open problem, and the future work names the breakout experiment that would study it directly. The benchmark uses seeded synthetic data throughout, so no personal data was processed and no human subjects were involved, and the one residual data protection concern, platform credentials in run artifacts, was answered with purging, rotation, and automatic redaction. Finally, the thesis documents its methods, audits, limitations, and null results in full, providing the transparency that evaluation work owes its readers. In terms of the United Nations Sustainable Development Goals, the work contributes to goal 9 on industry, innovation, and infrastructure, and to goal 12 on responsible consumption and production.

References

[1] L. Banh and G. Strobel, "Generative artificial intelligence," *Electronic Markets*, vol. 33, no. 1, p. 63, Dec. 2023. doi: 10.1007/s12525-023-00680-1. [Online]. Available: https://doi.org/10.1007/s12525-023-00680-1

[2] K. P. Murphy, *Machine learning: a probabilistic perspective*, 4th ed., ser. Adaptive computation and machine learning series. Cambridge, Mass.: MIT Press, 2013. ISBN 978-0-262-01802-9. [Online]. Available: https://mitpress.mit.edu/9780262018029/machine-learning/

[3] I. H. Sarker, "Machine Learning: Algorithms, Real-World Applications and Research Directions," *SN Computer Science*, vol. 2, no. 3, p. 160, May 2021. doi: 10.1007/s42979-021-00592-x. [Online]. Available: https://link.springer.com/10.1007/s42979-021-00592-x

[4] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, Nov. 2017. doi: 10.1109/MSP.2017.2743240. [Online]. Available: http://ieeexplore.ieee.org/document/8103164/

[5] P. Domingos, "A few useful things to know about machine learning," *Commun. ACM*, vol. 55, no. 10, pp. 78–87, Oct. 2012. doi: 10.1145/2347736.2347755. [Online]. Available: https://dl.acm.org/doi/10.1145/2347736.2347755

[6] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning*, ser. Springer Series in Statistics. New York, NY: Springer New York, 2009. ISBN 978-0-387-84857-0 978-0-387-84858-7. [Online]. Available: http://link.springer.com/10.1007/978-0-387-84858-7

[7] W. J. Murdoch, C. Singh, K. Kumbier, R. Abbasi-Asl, and B. Yu, "Definitions, methods, and applications in interpretable machine learning," *Proceedings of the National Academy of Sciences*, vol. 116, no. 44, pp. 22 071–22 080, Oct. 2019. doi: 10.1073/pnas.1900654116. [Online]. Available: https://pnas.org/doi/full/10.1073/pnas.1900654116

[8] D. De Silva and D. Alahakoon, "An artificial intelligence life cycle: From conception to production," *Patterns*, vol. 3, no. 6, p. 100489, Jun. 2022. doi: 10.1016/j.patter.2022.100489. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2666389922000745

[9] S. Raschka, J. Patterson, and C. Nolet, "Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence," *Information*, vol. 11, no. 4, p. 193, Apr. 2020. doi: 10.3390/info11040193. [Online]. Available: https://www.mdpi.com/2078-2489/11/4/193

[10] G. van Rossum, "Python Tutorial," Dec. 2003. [Online]. Available: https://worldcolleges.info/sites/default/files/tutorialPython_0.pdf

[11] E. S. Raymond, *The cathedral and the bazaar: musings on Linux and Open Source by an accidental revolutionary*. Beijing ; Cambridge, Mass: O'Reilly, 2001. ISBN 978-0-596-00131-5 978-0-596-00108-7

[12] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020. doi: 10.1038/s41586-020-2649-2. [Online]. Available: https://www.nature.com/articles/s41586-020-2649-2

[13] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: A system for large-scale machine learning," May 2016, arXiv:1605.08695 [cs]. [Online]. Available: http://arxiv.org/abs/1605.08695

[14] P. Barrett, J. Hunter, J. T. Miller, J.-C. Hsu, and P. Greenfield, "matplotlib – A Portable Python Plotting Package," in *ASP Conference Series*, vol. 347, Dec. 2005, p. 91, aDS Bibcode: 2005ASPC..347...91B. [Online]. Available: https://ui.adsabs.harvard.edu/abs/2005ASPC..347...91B

[15] A. Serban, K. Van Der Blom, H. Hoos, and J. Visser, "Adoption and Effects of Software Engineering Best Practices in Machine Learning," in *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. Bari Italy: ACM, Oct. 2020. doi: 10.1145/3382494.3410681. ISBN 978-1-4503-7580-1 pp. 1–12. [Online]. Available: https://dl.acm.org/doi/10.1145/3382494.3410681

[16] D. Kreuzberger, N. Kühl, and S. Hirschl, "Machine Learning Operations (MLOps): Overview, Definition, and Architecture," *IEEE Access*, vol. 11, pp. 31 866–31 879, 2023. doi: 10.1109/ACCESS.2023.3262138. [Online]. Available: https://ieeexplore.ieee.org/document/10081336

[17] D. Sculley, G. Holt, D. Golovin, E. Davydov, T. Phillips, D. Ebner, V. Chaudhary, M. Young, J.-F. Crespo, and D. Dennison, "Hidden technical debt in Machine learning systems," in *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'15, vol. 2. Cambridge, MA, USA: MIT Press, Dec. 2015, pp. 2503–2511.

[18] D. A. Tamburri, "Sustainable MLOps - Trends and Challenges," in *Proceedings - 2020 22nd International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*. IEEE, Sep. 2020. doi: 10.1109/SYNASC51798.2020.00015 pp. 17–23. [Online]. Available: https://research.tue.nl/en/publications/sustainable-mlops-trends-and-challenges/

[19] S. Mäkinen, H. Skogström, E. Laaksonen, and T. Mikkonen, "Who Needs MLOps: What Data Scientists Seek to Accomplish and How Can MLOps Help?" in *2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI (WAIN)*, May 2021. doi: 10.1109/WAIN52551.2021.00024 pp. 109–112. [Online]. Available: https://ieeexplore.ieee.org/document/9474355

[20] M. Testi, M. Ballabio, E. Frontoni, G. Iannello, S. Moccia, P. Soda, and G. Vessio, "MLOps: A Taxonomy and a Methodology," *IEEE Access*, vol. 10, pp. 63 606–63 618, 2022. doi: 10.1109/ACCESS.2022.3181730. [Online]. Available: https://ieeexplore.ieee.org/document/9792270/

[21] P. Ruf, M. Madan, C. Reich, and D. Ould-Abdeslam, "Demystifying MLOps and Presenting a Recipe for the Selection of Open-Source Tools," *Applied Sciences*, vol. 11, no. 19, p. 8861, Sep. 2021. doi: 10.3390/app11198861. [Online]. Available: https://www.mdpi.com/2076-3417/11/19/8861

[22] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, F. Xie, and C. Zumar, "Accelerating the Machine Learning Lifecycle with MLflow," *IEEE Data Engineering Bulletin*, vol. 41, no. 4, p. 7, Dec. 2018. [Online]. Available: https://people.eecs.berkeley.edu/~matei/papers/2018/ieee_mlflow.pdf

[23] "airflow," Apr. 2026, y. [Online]. Available: https://github.com/apache/airflow

[24] A. Gong, J. Campbell, and Great Expectations, "Great Expectations," Apr. 2026, y. [Online]. Available: https://github.com/great-expectations/great_expectations

[25] GitHub, Inc., "Build software better, together," Apr. 2026. [Online]. Available: https://github.com

[26] ——, "GitHub Actions," Apr. 2026. [Online]. Available: https://github.com/actions

[27] "git," Apr. 2026, y. [Online]. Available: https://github.com/git/git

[28] "kibana," Apr. 2026, y. [Online]. Available: https://github.com/elastic/kibana

[29] "grafana," Apr. 2026, y. [Online]. Available: https://github.com/grafana/grafana

[30] P. Singh, "Systematic review of data-centric approaches in artificial intelligence and machine learning," *Data Science and Management*, vol. 6, no. 3, pp. 144–157, Sep. 2023. doi: 10.1016/j.dsm.2023.06.001.

[Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2666764923000279

[31] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *J. Mach. Learn. Res.*, vol. 13, no. null, pp. 281–305, Feb. 2012. [Online]. Available: https://dl.acm.org/doi/10.5555/2188385.2188395

[32] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. [Online]. Available: https://www.deeplearningbook.org/

[33] A. E. Hoerl and R. W. Kennard, "Ridge Regression: Biased Estimation for Nonorthogonal Problems," *Technometrics*, vol. 42, no. 1, pp. 80–86, Feb. 2000. doi: 10.1080/00401706.2000.10485983. [Online]. Available: http://www.tandfonline.com/doi/abs/10.1080/00401706.2000.10485983

[34] N. Polyzotis, M. Zinkevich, S. Roy, E. Breck, and S. Whang, "Data Validation for Machine Learning," in *Proceedings of Machine Learning and Systems*, A. Talwalkar, V. Smith, and M. Zaharia, Eds., vol. 1, 2019, pp. 334–347. [Online]. Available: https://proceedings.mlsys.org/paper_files/paper/2019/file/928f1160e52192e3e0017fb63ab65391-Paper.pdf

[35] J. Gama, I. Žliobaitė, A. Bifet, M. Pechenizkiy, and A. Bouchachia, "A survey on concept drift adaptation," *ACM Computing Surveys (CSUR)*, vol. 46, no. 4, pp. 44:1–44:37, Mar. 2014. doi: 10.1145/2523813. [Online]. Available: https://dl.acm.org/doi/10.1145/2523813

[36] Jim Dowling, *Building Machine Learning Systems with a Feature Store*. O'Reilly Media, Inc, 2025, oCLC: 1425188410. [Online]. Available: https://www.oreilly.com/library/view/building-machine-learning/9781098165222/

[37] J. de la Rúa Martínez, F. Buso, A. Kouzoupis, A. A. Ormenisan, S. Niazi, D. Bzhalava, K. Mak, V. Jouffrey, M. Ronström, R. Cunningham, R. Zangis, D. Mukhedkar, A. Khazanchi, V. Vlassov, and J. Dowling, "The Hopsworks Feature Store for Machine Learning," in *Companion of the 2024 International Conference on Management of Data*, ser. SIGMOD '24. New York, NY, USA: Association for Computing Machinery, Jun. 2024. doi:

10.1145/3626246.3653389. ISBN 979-8-4007-0422-2 pp. 135–147. [Online]. Available: https://dl.acm.org/doi/10.1145/3626246.3653389

[38] J. Bloch, "How to Design a Good API and Why it Matters," in *Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*. ACM, 2006. doi: 10.1145/1176617.1176622 pp. 506–507.

[39] S. Fernandes and J. Bernardino, "What is BigQuery?" in *Proceedings of the 19th International Database Engineering & Applications Symposium*, ser. IDEAS '15. New York, NY, USA: Association for Computing Machinery, Jul. 2015. doi: 10.1145/2790755.2790797. ISBN 978-1-4503-3414-3 pp. 202–203. [Online]. Available: https://dl.acm.org/doi/10.1145/2790755.2790797

[40] "The Snowflake AI Data Cloud - Mobilize Data, Apps, and AI," 2026. [Online]. Available: https://www.snowflake.com/content/snowflake-site/global/en

[41] Amazon Web Services, Inc. or its affiliates, "S3 Getting Started," 2026. [Online]. Available: https://aws.amazon.com/s3/getting-started/

[42] S. Niazi, M. Ismail, S. Haridi, and J. Dowling, "HopsFS: Scaling Hierarchical File System Metadata Using NewSQL Databases," in *Proceedings of the 15th USENIX Conference on File and Storage Technologies (FAST '17)*. Berkeley, CA: USENIX Association, 2017. ISBN 978-1-931971-36-2 pp. 89–103, meeting Name: USENIX Conference on File and Storage Technologies. [Online]. Available: https://www.usenix.org/conference/fast17/technical-sessions/presentation/niazi

[43] "rondb," Mar. 2026, original-date: 2020-12-21T16:25:57Z. [Online]. Available: https://github.com/logicalclocks/rondb

[44] "OpenSearch," Mar. 2026, original-date: 2021-01-29T22:10:00Z. [Online]. Available: https://github.com/opensearch-project/OpenSearch

[45] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015. doi: 10.1038/nature14539. [Online]. Available: https://www.nature.com/articles/nature14539

[46] K. He, X. Zhang, S. Ren, and J. Sun, "Identity Mappings in Deep Residual Networks," Jul. 2016, arXiv:1603.05027 [cs.CV]. [Online]. Available: http://arxiv.org/abs/1603.05027

[47] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, Jan. 2015. doi: 10.1016/j.neunet.2014.09.003. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0893608014002135

[48] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, no. 6088, pp. 533–536, Oct. 1986. doi: 10.1038/323533a0. [Online]. Available: https://www.nature.com/articles/323533a0

[49] L. Alzubaidi, J. Zhang, A. J. Humaidi, A. Al-Dujaili, Y. Duan, O. Al-Shamma, J. Santamaría, M. A. Fadhel, M. Al-Amidie, and L. Farhan, "Review of deep learning: concepts, CNN architectures, challenges, applications, future directions," *Journal of Big Data*, vol. 8, no. 1, p. 53, Mar. 2021. doi: 10.1186/s40537-021-00444-8. [Online]. Available: https://journalofbigdata.springeropen.com/articles/10.1186/s40537-021-00444-8

[50] F. J. García Peñalvo and A. Vázquez Ingelmo, "What Do We Mean by GenAI? A Systematic Mapping of The Evolution, Trends, and Techniques Involved in Generative AI." *International Journal of Interactive Multimedia and Artificial Intelligence*, vol. 8, no. 4, pp. 7–16, Dec. 2023. doi: 10.9781/ijimai.2023.07.006. [Online]. Available: https://revistas.unir.net/index.php/ijimai/article/view/337

[51] P. Smolensky, "Information processing in dynamical systems: Foundations of harmony theory," *Parallel Distributed Process*, vol. 1, Jan. 1986.

[52] N. Metropolis, A. W. Rosenbluth, M. N. Rosenbluth, A. H. Teller, and E. Teller, "Equation of State Calculations by Fast Computing Machines," *The Journal of Chemical Physics*, vol. 21, no. 6, pp. 1087–1092, Jun. 1953. doi: 10.1063/1.1699114. [Online]. Available: https://pubs.aip.org/jcp/article/21/6/1087/202680/Equation-of-State-Calculations-by-Fast-Computing

[53] M. Trigka and E. Dritsas, "The Evolution of Generative AI: Trends and Applications," *IEEE Access*, vol. 13, pp. 98 504–98 529,

2025. doi: 10.1109/ACCESS.2025.3574660. [Online]. Available: https://ieeexplore.ieee.org/document/11016906/

[54] D. P. Kingma and M. Welling, "Auto-Encoding Variational Bayes," *CoRR*, vol. abs/1312.6114, 2013. [Online]. Available: https://api.semanticscholar.org/CorpusID:216078090

[55] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 2*, ser. NIPS'14. Cambridge, MA, USA: MIT Press, 2014, pp. 2672–2680.

[56] Y. Rubner, C. Tomasi, and L. J. Guibas, "The Earth Mover's Distance as a Metric for Image Retrieval," *International Journal of Computer Vision*, vol. 40, no. 2, pp. 99–121, Nov. 2000. doi: 10.1023/A:1026543900054. [Online]. Available: https://link.springer.com/10.1023/A:1026543900054

[57] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, ser. NIPS'17. Red Hook, NY, USA: Curran Associates Inc., Dec. 2017. ISBN 978-1-5108-6096-4 pp. 6000–6010. [Online]. Available: https://dl.acm.org/doi/10.5555/3295222.3295349

[58] Y. Huang, J. Luo, Y. Yu, Y. Zhang, F. Lei, Y. Wei, S. He, L. Huang, X. Liu, J. Zhao, and K. Liu, "DA-Code: Agent Data Science Code Generation Benchmark for Large Language Models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024. doi: 10.18653/v1/2024.emnlp-main.748 pp. 13 487–13 521. [Online]. Available: https://aclanthology.org/2024.emnlp-main.748/

[59] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding," May 2019, arXiv:1810.04805 [cs]. [Online]. Available: http://arxiv.org/abs/1810.04805

[60] A. Radford and K. Narasimhan, "Improving Language Understanding by Generative Pre-Training," 2018. [Online]. Available: https://api.semanticscholar.org/CorpusID:49313245

[61] C. Raffel, N. Shazeer, A. Roberts, K. Lee, S. Narang, M. Matena, Y. Zhou, W. Li, and P. J. Liu, "Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer," Sep. 2023, arXiv:1910.10683 [cs]. [Online]. Available: http://arxiv.org/abs/1910.10683

[62] A. Bandi, P. V. S. R. Adapa, and Y. E. V. P. K. Kuchi, "The Power of Generative AI: A Review of Requirements, Models, Input–Output Formats, Evaluation Metrics, and Challenges," *Future Internet*, vol. 15, no. 8, p. 260, Jul. 2023. doi: 10.3390/fi15080260. [Online]. Available: https://www.mdpi.com/1999-5903/15/8/260

[63] M. A. K. Raiaan, M. S. H. Mukta, K. Fatema, N. M. Fahad, S. Sakib, M. M. J. Mim, J. Ahmad, M. E. Ali, and S. Azam, "A Review on Large Language Models: Architectures, Applications, Taxonomies, Open Issues and Challenges," *IEEE Access*, vol. 12, pp. 26 839–26 874, 2024. doi: 10.1109/ACCESS.2024.3365742. [Online]. Available: https://ieeexplore.ieee.org/document/10433480/

[64] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, S. Agarwal, A. Herbert-Voss, G. Krueger, T. Henighan, R. Child, A. Ramesh, D. Ziegler, J. Wu, C. Winter, C. Hesse, M. Chen, E. Sigler, M. Litwin, S. Gray, B. Chess, J. Clark, C. Berner, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei, "Language Models are Few-Shot Learners," in *Advances in Neural Information Processing Systems*, H. Larochelle, M. Ranzato, R. Hadsell, M. F. Balcan, and H. Lin, Eds., vol. 33. Curran Associates, Inc., 2020, pp. 1877–1901. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2020/file/1457c0d6bfcb4967418bfb8ac142f64a-Paper.pdf

[65] L. Ouyang, J. Wu, X. Jiang, D. Almeida, C. L. Wainwright, P. Mishkin, C. Zhang, S. Agarwal, K. Slama, A. Ray, J. Schulman, J. Hilton, F. Kelton, L. Miller, M. Simens, A. Askell, P. Welinder, P. Christiano, J. Leike, and R. Lowe, "Training language models to follow instructions with human feedback," Mar. 2022, arXiv:2203.02155 [cs.CL]. [Online]. Available: http://arxiv.org/abs/2203.02155

[66] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal Policy Optimization Algorithms," *CoRR*, vol. abs/1707.06347, 2017, arXiv: 1707.06347. [Online]. Available: http://arxiv.org/abs/1707.06347

[67] S. Yin, C. Fu, S. Zhao, K. Li, X. Sun, T. Xu, and E. Chen, "A survey on multimodal large language models," *National Science Review*, vol. 11, no. 12, p. nwae403, Nov. 2024. doi: 10.1093/nsr/nwae403. [Online]. Available: https://academic.oup.com/nsr/article/doi/10.1093/nsr/nwae403/7896414

[68] N. Ratner, Y. Levine, Y. Belinkov, O. Ram, I. Magar, O. Abend, E. Karpas, A. Shashua, K. Leyton-Brown, and Y. Shoham, "Parallel Context Windows for Large Language Models," in *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, A. Rogers, J. Boyd-Graber, and N. Okazaki, Eds. Toronto, Canada: Association for Computational Linguistics, Jul. 2023. doi: 10.18653/v1/2023.acl-long.352 pp. 6383–6402. [Online]. Available: https://aclanthology.org/2023.acl-long.352/

[69] K. Hong, A. Troynikov, and J. Huber, "Context Rot: How Increasing Input Tokens Impacts LLM Performance," Chroma, Tech. Rep., Jul. 2025. [Online]. Available: https://trychroma.com/research/context-rot

[70] O. E. Cinar and N. H. Guldemir, "From Prompt Engineering to Context Engineering: A Comparative Analysis of AI Interaction Paradigms for Large Language Models," in *2026 5th International Informatics and Software Engineering Conference (IISEC)*, Feb. 2026. doi: 10.1109/IISEC69317.2026.11418464 pp. 104–109. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/11418464

[71] T. Wu, "Improving General-Purpose LLM Models for Power Cable Tasks Using Structured Context Engineering," in *2025 7th International Conference on Smart Power & Internet Energy Systems (SPIES)*, Oct. 2025. doi: 10.1109/SPIES67451.2025.11381468 pp. 41–45. [Online]. Available: https://ieeexplore.ieee.org/document/11381468

[72] Z. Gao, L. Tong, and Z. Zhang, "Detecting and Evaluating Bias in Large Language Models: Concepts, Methods, and Challenges," *Journal of Behavioral Data Science*, vol. 6, no. 1, pp. 1–68,

Feb. 2026. doi: 10.35566/jbds/gao. [Online]. Available: https://jbds.isdsa.org/jbds/article/view/182

[73] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large Language Models for Software Engineering: A Systematic Literature Review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, Nov. 2024. doi: 10.1145/3695988. [Online]. Available: https://dl.acm.org/doi/10.1145/3695988

[74] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. R. Narasimhan, "SWE-bench: Can Language Models Resolve Real-world Github Issues?" in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=VTF8yNQM66

[75] Anthropic PBC, "System Card: Claude Opus 4.6," Feb. 2026. [Online]. Available: https://www-cdn.anthropic.com/14e4fb01875d2a69f646fa5e574dea2b1c0ff7b5.pdf

[76] OpenAI, "Introducing GPT-5.2," Dec. 2025. [Online]. Available: https://openai.com/index/introducing-gpt-5-2/

[77] F. Tambon, A. Moradi-Dakhel, A. Nikanjam, F. Khomh, M. C. Desmarais, and G. Antoniol, "Bugs in large language models generated code: an empirical study," *Empirical Software Engineering*, vol. 30, no. 3, p. 65, May 2025. doi: 10.1007/s10664-025-10614-4. [Online]. Available: https://link.springer.com/10.1007/s10664-025-10614-4

[78] X. Ma, X. Du, C. Li, J. Meng, X. Fang, W. Ding, and Z. Zheng, "How Do Large Language Models Perform in Deep Learning Code Generation? An Empirical Study," *ACM Transactions on Software Engineering and Methodology*, p. 3816024, May 2026. doi: 10.1145/3816024. [Online]. Available: https://dl.acm.org/doi/10.1145/3816024

[79] J. Liu, S. Xu, Y. Yao, Y. Shen, Y. Zhao, X. Zhu, P. Yu, F. Xu, and X. Ma, "Robustness evaluation and enhancement of LLMs in code generation: an empirical study," *Empirical Software Engineering*, vol. 31, no. 4, p. 112, Apr. 2026. doi: 10.1007/s10664-026-10856-w. [Online]. Available: https://doi.org/10.1007/s10664-026-10856-w

[80] A. Karpathy, “There’s a new kind of coding I call "vibe coding", where you fully give in to the vibes, embrace exponentials, and forget that the code even exists. It’s possible because the LLMs (e.g. Cursor Composer w Sonnet) are getting too good. Also I just talk to Composer with SuperWhisper,” Feb. 2025. [Online]. Available: https://x.com/karpathy/status/1886192184808149383

[81] N. Beaulieu, S. Dascalu, and E. Hand, “Vibe Coding: A Multivocal Systematic Mapping Study,” in *2025 IEEE/ACIS 29th International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, Jun. 2025. doi: 10.1109/SNPD65828.2025.11254176. ISSN 2693-8421 pp. 266–273, iSSN: 2693-8421. [Online]. Available: https://ieeexplore.ieee.org/document/11254176

[82] Akhilesh Gadde, “Democratizing Software Engineering through Generative AI and Vibe Coding: The Evolution of No-Code Development,” *Journal of Computer Science and Technology Studies*, vol. 7, no. 4, pp. 556–572, May 2025. doi: 10.32996/jcsts.2025.7.4.66. [Online]. Available: https://al-kindipublisher.com/index.php/jcsts/article/view/9582

[83] X. He, K. Zhao, and X. Chu, “AutoML: A survey of the state-of-the-art,” *Knowledge-Based Systems*, vol. 212, p. 106622, Jan. 2021. doi: 10.1016/j.knosys.2020.106622. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0950705120307516

[84] L. Prechelt, “Early Stopping — But When?” in *Neural Networks: Tricks of the Trade*, G. Montavon, G. B. Orr, and K.-R. Müller, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 7700, pp. 53–67. ISBN 978-3-642-35288-1 978-3-642-35289-8 Series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-642-35289-8_5

[85] N. V. Queipo, R. T. Haftka, W. Shyy, T. Goel, R. Vaidyanathan, and P. Kevin Tucker, “Surrogate-based analysis and optimization,” *Progress in Aerospace Sciences*, vol. 41, no. 1, pp. 1–28, Jan. 2005. doi: 10.1016/j.paerosci.2005.02.001. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0376042105000102

[86] H. Pham, M. Y. Guan, B. Zoph, Q. V. Le, and J. Dean, “Efficient Neural Architecture Search via Parameter Sharing,” Feb. 2018,

arXiv:1802.03268 [cs.LG]. [Online]. Available: http://arxiv.org/abs/1802.03268

[87] H. Choi, J. Moran, N. Matsumoto, M. E. Hernandez, and J. H. Moore, "Aliro: an automated machine learning tool leveraging large language models," *Bioinformatics*, vol. 39, no. 10, p. btad606, Oct. 2023. doi: 10.1093/bioinformatics/btad606. [Online]. Available: https://doi.org/10.1093/bioinformatics/btad606

[88] L. Breiman, "Random Forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct. 2001. doi: 10.1023/A:1010933404324. [Online]. Available: https://link.springer.com/10.1023/A:1010933404324

[89] S. Chen, W. Zhai, C. Chai, and X. Shi, "LLM2AutoML: Zero-Code AutoML Framework Leveraging Large Language Models," in *2024 International Conference on Intelligent Robotics and Automatic Control (IRAC)*, Nov. 2024. doi: 10.1109/IRAC63143.2024.10871761 pp. 285–290. [Online]. Available: https://ieeexplore.ieee.org/document/10871761

[90] I. Guyon, J. Weston, S. Barnhill, and V. Vapnik, "Gene Selection for Cancer Classification using Support Vector Machines," *Machine Learning*, vol. 46, no. 1, pp. 389–422, 2002. doi: 10.1023/A:1012487302797

[91] I. Guyon and A. Elisseeff, "An Introduction to Variable and Feature Selection," *Journal of Machine Learning Research*, vol. 3, pp. 1157–1182, 2003.

[92] Y. Gu, H. You, J. Cao, M. Yu, H. Fan, and S. Qian, "Large Language Models for Constructing and Optimizing Machine Learning Workflows: A Survey," *ACM Trans. Softw. Eng. Methodol.*, Oct. 2025. doi: 10.1145/3773084 Just Accepted. [Online]. Available: https://dl.acm.org/doi/10.1145/3773084

[93] P. Hart, N. Nilsson, and B. Raphael, "A Formal Basis for the Heuristic Determination of Minimum Cost Paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968. doi: 10.1109/TSSC.1968.300136. [Online]. Available: http://ieeexplore.ieee.org/document/4082128/

[94] R. Hai, C. Koutras, C. Quix, and M. Jarke, "Data Lakes: A Survey of Functions and Systems," *IEEE Transactions on Knowledge and Data Engineering*, vol. 35, no. 12, pp. 12 571–12 590, 2023. doi: 10.1109/TKDE.2023.3270101. [Online]. Available: https://ieeexplore.ieee.org/document/10107808/

[95] R. Dale, "Sovereign AI in 2025," *Natural Language Processing*, vol. 31, no. 5, pp. 1312–1321, Sep. 2025. doi: 10.1017/nlp.2025.10007. [Online]. Available: https://www.cambridge.org/core/product/identifier/S2977042425100071/type/journal_article

[96] A. Turchin, D. Denkenberger, and B. P. Green, "Global Solutions vs. Local Solutions for the AI Safety Problem," *Big Data and Cognitive Computing*, vol. 3, no. 1, p. 16, Feb. 2019. doi: 10.3390/bdcc3010016. [Online]. Available: https://www.mdpi.com/2504-2289/3/1/16

[97] Z. Wang, "Generative AI-Making and State-Making: Sovereign AI Race and the Future of Digital Geopolitics," *Politics and Governance*, vol. 13, p. 10222, Nov. 2025. doi: 10.17645/pag.10222. [Online]. Available: https://www.cogitatiopress.com/politicsandgovernance/article/view/10222

[98] Y. Sun, M. Bergen, and B. Berthelot, "France's Mistral in Funding Talks at About €20 Billion Valuation," *Bloomberg.com*, Jun. 2026. [Online]. Available: https://www.bloomberg.com/news/articles/2026-06-12/france-s-mistral-in-funding-talks-at-about-20-billion-valuation

[99] G. Chemla, T. Quddus Islam, and S. Li, "Strategic Requirements for a Sovereign AI Infrastructure," in *NATO Science for Peace and Security Series - E: Human and Societal Dynamics*, F. Andrés Pérez and R. García Alonso, Eds. IOS Press, May 2026. ISBN 978-1-64368-658-5. [Online]. Available: https://ebooks.iospress.nl/doi/10.3233/NHSDP260040

[100] A. S. Rao and M. P. Georgeff, "Modeling rational agents within a BDI-architecture," in *Proceedings of the Second International Conference on Principles of Knowledge Representation and Reasoning*, ser. KR'91. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., Apr. 1991. ISBN 978-1-55860-165-9 pp. 473–484. [Online]. Available: https://dl.acm.org/doi/10.5555/3087158.3087205

[101] M. Wooldridge and N. R. Jennings, "Intelligent agents: theory and practice," *The Knowledge Engineering Review*, vol. 10, no. 2, pp. 115–152, Jun. 1995. doi: 10.1017/S0269888900008122. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0269888900008122/type/journal_article

[102] N. R. Jennings and M. Wooldridge, "Applications of Intelligent Agents," in *Agent Technology: Foundations, Applications, and Markets*, N. R. Jennings and M. J. Wooldridge, Eds. Berlin, Heidelberg: Springer, 1998, pp. 3–28. ISBN 978-3-662-03678-5. [Online]. Available: https://doi.org/10.1007/978-3-662-03678-5_1

[103] L. Panait and S. Luke, "Cooperative Multi-Agent Learning: The State of the Art," *Autonomous Agents and Multi-Agent Systems*, vol. 11, no. 3, pp. 387–434, Nov. 2005. doi: 10.1007/s10458-005-2631-2. [Online]. Available: http://link.springer.com/10.1007/s10458-005-2631-2

[104] M. J. Wooldridge, *An introduction to multiagent systems*, 2nd ed. Chichester: Wiley, 2012. ISBN 978-0-470-51946-2. [Online]. Available: https://uranos.ch/research/references/Wooldridge_2001/TLTK.pdf

[105] N. R. Jennings, "Commitments and conventions: The foundation of coordination in multi-agent systems," *The Knowledge Engineering Review*, vol. 8, no. 3, pp. 223–250, Sep. 1993. doi: 10.1017/S0269888900000205. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0269888900000205/type/journal_article

[106] B. J. Grosz and S. Kraus, "Collaborative plans for complex group action," *Artificial Intelligence*, vol. 86, no. 2, pp. 269–357, Oct. 1996. doi: 10.1016/0004-3702(95)00103-4. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/0004370295001034

[107] J. M. Bradshaw, "An introduction to software agents," in *Software Agents*. Cambridge, MA, USA: MIT Press, 1997, pp. 3–46. ISBN 0-262-52234-9. [Online]. Available: https://dl.acm.org/doi/10.5555/267985.267993

[108] S. Franklin and A. Graesser, "Is It an agent, or just a program?: A taxonomy for autonomous agents," in *Intelligent Agents III Agent Theories, Architectures, and Languages*, J. P. Müller, M. J. Wooldridge,

and N. R. Jennings, Eds. Berlin, Heidelberg: Springer, 1997. doi: 10.1007/BFb0013570. ISBN 978-3-540-68057-4 pp. 21–35. [Online]. Available: https://link.springer.com/chapter/10.1007/BFb0013570

[109] H. S. Nwana, "Software agents: an overview," *The Knowledge Engineering Review*, vol. 11, no. 3, pp. 205–244, Sep. 1996. doi: 10.1017/S026988890000789X. [Online]. Available: https://www.cambridge.org/core/product/identifier/S026988890000789X/type/journal_article

[110] N. R. Jennings, "On agent-based software engineering," *Artificial Intelligence*, vol. 117, no. 2, pp. 277–296, Mar. 2000. doi: 10.1016/S0004-3702(99)00107-1. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0004370299001071

[111] M. R. Genesereth and S. P. Ketchpel, "Software agents," *Communications of the ACM*, vol. 37, no. 7, p. 48, Jul. 1994. doi: 10.1145/176789.176794. [Online]. Available: https://dl.acm.org/doi/10.1145/176789.176794

[112] L. Wang, C. Ma, X. Feng, Z. Zhang, H. Yang, J. Zhang, Z. Chen, J. Tang, X. Chen, Y. Lin, W. X. Zhao, Z. Wei, and J. Wen, "A survey on large language model based autonomous agents," *Frontiers of Computer Science*, vol. 18, no. 6, p. 186345, Dec. 2024. doi: 10.1007/s11704-024-40231-1. [Online]. Available: https://link.springer.com/10.1007/s11704-024-40231-1

[113] J. Liu, K. Wang, Y. Chen, X. Peng, Z. Chen, L. Zhang, and Y. Lou, "Large Language Model-Based Agents for Software Engineering: A Survey," *ACM Transactions on Software Engineering and Methodology*, p. 3796507, Mar. 2026. doi: 10.1145/3796507. [Online]. Available: https://dl.acm.org/doi/10.1145/3796507

[114] C. Meske, T. Hermanns, E. Von der Weiden, K.-U. Loser, and T. Berger, "Vibe Coding as a Reconfiguration of Intent Mediation in Software Development: Definition, Implications, and Research Agenda," *IEEE Access*, vol. 13, pp. 213 242–213 259, 2025. doi: 10.1109/ACCESS.2025.3645466. [Online]. Available: https://ieeexplore.ieee.org/document/11303023

[115] Anthropic PBC, "Claude Code overview," 2026. [Online]. Available: https://code.claude.com/docs/en/overview

[116] "GitHub Copilot," 2026. [Online]. Available: https://github.com/copilot

[117] Anomaly Innovations Inc., "anomalyco/opencode," Apr. 2026, original-date: 2025-04-30T20:08:00Z. [Online]. Available: https://github.com/anomalyco/opencode

[118] M. Puvvadi, S. K. Arava, A. Santoria, S. S. P. Chennupati, and H. V. Puvvadi, "Coding Agents: A Comprehensive Survey of Automated Bug Fixing Systems and Benchmarks," in *2025 IEEE 14th International Conference on Communication Systems and Network Technologies (CSNT)*, 2025. doi: 10.1109/CSNT64827.2025.10968728 pp. 680–686. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/10968728

[119] V. Tawosi, S. Alamir, X. Liu, and M. Veloso, "LLM Agents for Automated Dependency Upgrades," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, Nov. 2025. doi: 10.1109/ASEW67777.2025.00016. ISSN 2151-0849 pp. 34–37, iSSN: 2151-0849. [Online]. Available: https://ieeexplore.ieee.org/document/11334457

[120] M. Watanabe, H. Li, Y. Kashiwa, B. Reid, H. Iida, and A. E. Hassan, "On the Use of Agentic Coding: An Empirical Study of Pull Requests on GitHub," *ACM Transactions on Software Engineering and Methodology*, p. 3798166, Mar. 2026. doi: 10.1145/3798166. [Online]. Available: https://dl.acm.org/doi/10.1145/3798166

[121] U. Yusuf, G. Caumartin, and D. E. Costa, "Beyond More Context: How Granularity and Order Drive Code Completion Quality," in *2025 40th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW)*, Nov. 2025. doi: 10.1109/ASEW67777.2025.00075. ISSN 2151-0849 pp. 371–374, iSSN: 2151-0849. [Online]. Available: https://ieeexplore.ieee.org/document/11334377

[122] W. Fan, Y. Ding, L. Ning, S. Wang, H. Li, D. Yin, T.-S. Chua, and Q. Li, "A Survey on RAG Meeting LLMs: Towards Retrieval-Augmented Large Language Models," in *Proceedings of the 30th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*.

Barcelona Spain: ACM, Aug. 2024. doi: 10.1145/3637528.3671470. ISBN 979-8-4007-0490-1 pp. 6491–6501. [Online]. Available: https://dl.acm.org/doi/10.1145/3637528.3671470

[123] M. C. Ramos, C. J. Collison, and A. D. White, "A review of large language models and autonomous agents in chemistry," *Chemical Science*, vol. 16, no. 6, pp. 2514–2572, 2025. doi: 10.1039/D4SC03921A. [Online]. Available: https://xlink.rsc.org/?DOI=D4SC03921A

[124] K. Shuster, S. Poff, M. Chen, D. Kiela, and J. Weston, "Retrieval Augmentation Reduces Hallucination in Conversation," in *Findings of the Association for Computational Linguistics: EMNLP 2021*, M.-F. Moens, X. Huang, L. Specia, and S. W.-t. Yih, Eds. Punta Cana, Dominican Republic: Association for Computational Linguistics, Nov. 2021. doi: 10.18653/v1/2021.findings-emnlp.320 pp. 3784–3803. [Online]. Available: https://aclanthology.org/2021.findings-emnlp.320/

[125] L. Huang, W. Yu, W. Ma, W. Zhong, Z. Feng, H. Wang, Q. Chen, W. Peng, X. Feng, B. Qin, and T. Liu, "A Survey on Hallucination in Large Language Models: Principles, Taxonomy, Challenges, and Open Questions," *ACM Transactions on Information Systems*, vol. 43, no. 2, pp. 1–55, Mar. 2025. doi: 10.1145/3703155. [Online]. Available: https://dl.acm.org/doi/10.1145/3703155

[126] P. Chhikara, D. Khant, S. Aryan, T. Singh, and D. Yadav, "Mem0: Building Production-Ready AI Agents with Scalable Long-Term Memory," Apr. 2025, arXiv:2504.19413 [cs]. [Online]. Available: http://arxiv.org/abs/2504.19413

[127] T. Schick, J. Dwivedi-Yu, R. Dessí, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: language models can teach themselves to use tools," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2023. [Online]. Available: https://dl.acm.org/doi/10.5555/3666122.3669119

[128] M. Zhong, P. Liu, Y. Chen, D. Wang, X. Qiu, and X. Huang, "Extractive Summarization as Text Matching," in *Proceedings of the 58th Annual Meeting of the Association for Computational*

Linguistics, D. Jurafsky, J. Chai, N. Schluter, and J. Tetreault, Eds. Online: Association for Computational Linguistics, Jul. 2020. doi: 10.18653/v1/2020.acl-main.552 pp. 6197–6208. [Online]. Available: https://aclanthology.org/2020.acl-main.552/

[129] J. Gonzalez, S. Patil, X. Wang, and T. Zhang, "Gorilla: Large Language Model Connected with Massive APIs," in *Advances in Neural Information Processing Systems 37*. Vancouver, BC, Canada: Neural Information Processing Systems Foundation, Inc. (NeurIPS), 2024. doi: 10.52202/079017-4020. ISBN 979-8-3313-1438-5 pp. 126 544–126 565. [Online]. Available: http://www.proceedings.com/079017-4020.html

[130] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian, S. Zhao, L. Hong, R. Tian, R. Xie, J. Zhou, M. Gerstein, d. li, Z. Liu, and M. Sun, "ToolLLM: Facilitating Large Language Models to Master 16000+ Real-world APIs," in *The Twelfth International Conference on Learning Representations*, 2024. [Online]. Available: https://openreview.net/forum?id=dHng2O0Jjr

[131] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[132] T. Berners-Lee, "Information Management: A Proposal," May 1990. [Online]. Available: https://www.w3.org/History/1989/proposal-msw.html

[133] D. Crockford, "The application/json Media Type for JavaScript Object Notation (JSON)," Internet Engineering Task Force, Request for Comments RFC 4627, Jul. 2006, num Pages: 10. [Online]. Available: https://datatracker.ietf.org/doc/rfc4627

[134] S. G. Patil, H. Mao, F. Yan, C. C.-J. Ji, V. Suresh, I. Stoica, and J. E. Gonzalez, "The Berkeley Function Calling Leaderboard (BFCL): From Tool Use to Agentic Evaluation of Large Language Models," in *Forty-second International Conference on Machine Learning*, 2025. [Online]. Available: https://openreview.net/forum?id=2GmDdhBdDk

[135] D. E. Perry and A. L. Wolf, "Foundations for the study of software architecture," *ACM SIGSOFT Software Engineering Notes*, vol. 17,

no. 4, pp. 40–52, Oct. 1992. doi: 10.1145/141874.141884. [Online]. Available: https://dl.acm.org/doi/10.1145/141874.141884

[136] M. Wang, Y. Zhang, B. Yu, B. Hao, C. Peng, Y. Chen, W. Zhou, J. Gu, C. Zhuang, R. Guo, W. Wang, and X. Zhao, "Function Calling in Large Language Models: Industrial Practices, Challenges, and Future Directions," *ACM Computing Surveys*, vol. 58, no. 9, pp. 1–37, Jul. 2026. doi: 10.1145/3788284. [Online]. Available: https://dl.acm.org/doi/10.1145/3788284

[137] M. Henning, "API design matters," *Communications of the ACM*, vol. 52, no. 5, pp. 46–56, 2009. doi: 10.1145/1506409.1506424

[138] F. J. Corbató, M. Merwin-Daggett, and R. C. Daley, "An experimental time-sharing system," in *Proceedings of the May 1-3, 1962, spring joint computer conference on - AIEE-IRE '62 (Spring)*. San Francisco, California: ACM Press, 1962. doi: 10.1145/1460833.1460871 p. 335. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1460833.1460871

[139] D. Wixon, J. Whiteside, M. Good, and S. Jones, "Building a user-defined interface," in *Proceedings of the SIGCHI conference on Human Factors in Computing Systems - CHI '83*. Boston, Massachusetts, United States: ACM Press, 1983. doi: 10.1145/800045.801574. ISBN 978-0-89791-121-4 pp. 24–27. [Online]. Available: http://portal.acm.org/citation.cfm?doid=800045.801574

[140] D. M. Ritchie and K. Thompson, "The UNIX time-sharing system," *Communications of the ACM*, vol. 17, no. 7, pp. 365–375, Jul. 1974. doi: 10.1145/361011.361061. [Online]. Available: https://dl.acm.org/doi/10.1145/361011.361061

[141] S. R. Bourne, "An Introduction to the UNIX Shell," Murray Hill, New Jersey, United States of America, Nov. 1997. [Online]. Available: https://cscie26.dce.harvard.edu/~dce-lib113/reference/unix/bourne_shell.pdf

[142] C. Ramey, "What's GNU: Bash - The GNU Shell," Jun. 1994. [Online]. Available: https://www.linuxjournal.com/article/2784

[143] C. Ramsey and B. Fox, "The GNU Bash Reference Manual, for Bash," May 2025. [Online]. Available: https://www.gnu.org/software/bash/manual/bash.pdf

[144] M. Agarwal, J. J. Barroso, T. Chakraborti, E. M. Dow, K. Fadnis, B. Godoy, M. Pallan, and K. Talamadupula, "Project CLAI: Instrumenting the Command Line as a New Environment for AI Agents," 2020, version Number: 2. [Online]. Available: https://arxiv.org/abs/2002.00762

[145] M. Agarwal, T. Chakraborti, Q. Fu, D. Gros, X. V. Lin, J. Maene, K. Talamadupula, Z. Teng, and J. White, "NeurIPS 2020 NLC2CMD Competition: Translating Natural Language to Bash Commands," in *Proceedings of the NeurIPS 2020 Competition and Demonstration Track*, ser. Proceedings of Machine Learning Research, H. J. Escalante and K. Hofmann, Eds., vol. 133. PMLR, Dec. 2021, pp. 302–324. [Online]. Available: https://proceedings.mlr.press/v133/agarwal21b.html

[146] F. Westenfelder, E. Hemberg, S. Moskal, U.-M. O'Reilly, and S. Chiricescu, "LLM-Supported Natural Language to Bash Translation," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*. Albuquerque, New Mexico: Association for Computational Linguistics, 2025. doi: 10.18653/v1/2025.naacl-long.555 pp. 11 135–11 147. [Online]. Available: https://aclanthology.org/2025.naacl-long.555

[147] X. Hou, Y. Zhao, S. Wang, and H. Wang, "Model Context Protocol (MCP): Landscape, Security Threats, and Future Research Directions," *ACM Trans. Softw. Eng. Methodol.*, Feb. 2026. doi: 10.1145/3796519 Just Accepted. [Online]. Available: https://dl.acm.org/doi/10.1145/3796519

[148] N. Gunasinghe and N. Marcus, *Language Server Protocol and Implementation: Supporting Language-Smart Editing and Programming Tools*. Apress, 2021. ISBN 978-1-4842-7791-1. [Online]. Available: https://books.google.se/books?id=zeuezgEACAAJ

[149] D. Delimarsky, "MCP vs. CLI," Mar. 2026. [Online]. Available: https://www.youtube.com/watch?v=LK19nAd7a1E

[150] D. Soria Parra, "Design Principles," 2026. [Online]. Available: https://modelcontextprotocol.io/community/design-principles

[151] J. E. Arango Ossa, D. Domenico, A. P. Steinberg, E. Havasov, G. Gundem, K. Liosis, A. Grande, J. Gutierrez-Abril, E. Papaemmanuil, S. Shah, and A. W. McPherson, "Abstract 27: An MCP-enabled AI agent for automated multimodal genomics analysis in the Isabl Platform." *Cancer Research*, vol. 86, no. 7_Supplement, pp. 27–27, Apr. 2026. doi: 10.1158/1538-7445.AM2026-27. [Online]. Available: https://aacrjournals.org/cancerres/article/86/7_Supplement/27/775776/Abstract-27-An-MCP-enabled-AI-agent-for-automated

[152] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "ReAct: Synergizing Reasoning and Acting in Language Models," 2022, version Number: 3. [Online]. Available: https://arxiv.org/abs/2210.03629

[153] Slack Technologies, LLC, a Salesforce company, "Slack | AI Work Platform & Productivity Tools," 2026. [Online]. Available: https://slack.com

[154] Blender Foundation, "Blender - The Free and Open Source 3D Creation Software — blender.org," 2026. [Online]. Available: https://www.blender.org/

[155] I. Zuykov, "Architectural Principles for Multi-Agent Systems Based on The Model Context Protocol," *Emerging Frontiers Library for The American Journal of Engineering and Technology*, vol. 8, no. 03, pp. 158–169, Mar. 2026. [Online]. Available: https://emergingsociety.org/index.php/efltajet/article/view/1233

[156] A. Z. Liu, J. Choi, S. Sohn, Y. Fu, J. Kim, D.-K. Kim, X. Wang, J. Yoo, and H. Lee, "SkillAct: Using Skill Abstractions Improves LLM Agents," in *ICML 2024 Workshop on LLMs and Cognition*, 2024. [Online]. Available: https://openreview.net/forum?id=6LG3cIRrF4

[157] M. Shridhar, X. Yuan, M.-A. Cote, Y. Bisk, A. Trischler, and M. Hausknecht, "{ALFW}orld: Aligning Text and Embodied Environments for Interactive Learning," in *International Conference on Learning Representations*, 2021. [Online]. Available: https://openreview.net/forum?id=0IOX0YcCdTn

[158] J. Gruber, "Daring Fireball: Markdown," Dec. 2004. [Online]. Available: https://daringfireball.net/projects/markdown/

[159] W. Chatlatanagulchai, K. Thonglek, B. Reid, Y. Kashiwa, P. Leelaprute, A. Rungsawang, B. Manaskasemsak, and H. Iida, "On the Use of Agentic Coding Manifests: An Empirical Study of Claude Code," in *Product-Focused Software Process Improvement*, G. Scanniello, V. Lenarduzzi, S. Romano, S. Vegas, and R. Francese, Eds. Cham: Springer Nature Switzerland, 2026, vol. 16361, pp. 543–551. ISBN 978-3-032-12088-5 978-3-032-12089-2 Series Title: Lecture Notes in Computer Science. [Online]. Available: https://link.springer.com/10.1007/978-3-032-12089-2_40

[160] Anthropic PBC, "How Claude remembers your project," 2026. [Online]. Available: https://code.claude.com/docs/en/memory

[161] "agents.md," Apr. 2026, original-date: 2025-08-19T17:22:54Z. [Online]. Available: https://github.com/agentsmd/agents.md

[162] J. Johnson, T. L. Roberts, W. Verplank, D. C. Smith, C. H. Irby, M. Beard, and K. Mackey, "The Xerox Star: a retrospective," *Computer*, vol. 22, no. 9, pp. 11–26, Sep. 1989. doi: 10.1109/2.35211. [Online]. Available: https://doi.org/10.1109/2.35211

[163] Anthropic PBC, "Equipping agents for the real world with Agent Skills," Oct. 2025. [Online]. Available: https://www.anthropic.com/engineering/equipping-agents-for-the-real-world-with-agent-skills

[164] S. Willison, "Code execution with MCP: Building more efficient agents," Nov. 2025. [Online]. Available: https://simonwillison.net/2025/Nov/4/code-execution-with-mcp/

[165] musistudio, "Progressive Disclosure of Agent Tools from the Perspective of CLI Tool Style," Jan. 2026. [Online]. Available: https://github.com/musistudio/claude-code-router/blob/main/blog/en/progressive-disclosure-of-agent-tools-from-the-perspective-of-cli-tool-style.md

[166] D. Schonholtz, "agent-harness 0.1.0," Aug. 2023. [Online]. Available: https://pypi.org/project/agent-harness/0.1.0/

[167] A. Bigeard, L. Nashold, R. Krishnan, and S. Wu, "Finance Agent Benchmark: Benchmarking LLMs on Real-world Financial Research Tasks," May 2025. [Online]. Available: https://arxiv.org/abs/2508.00828

[168] M. Hashimoto, "My AI Adoption Journey," Feb. 2026. [Online]. Available: https://mitchellh.com/writing/my-ai-adoption-journey

[169] OpenAI, "Harness engineering: leveraging Codex in an agent-first world," Feb. 2026. [Online]. Available: https://openai.com/index/harness-engineering/

[170] V. Trivedy, "The Anatomy of an Agent Harness," Mar. 2026. [Online]. Available: https://www.langchain.com/blog/the-anatomy-of-an-agent-harness

[171] M. Zaharia, K. Uhlenhuth, and C. Zumar, "Introducing Omnigent: A Meta-Harness to Combine, Control and Share Your Agents," Jun. 2026. [Online]. Available: https://www.databricks.com/blog/introducing-omnigent-meta-harness-combine-control-and-share-your-agents

[172] J. Ball, "Altimeter First Pass: A conversation with Matei Zaharia from Databricks about Omnigent," Jul. 2026. [Online]. Available: https://www.linkedin.com/posts/jamin-ball-49366137_next-up-on-altimeter-first-pass-a-conversation-ugcPost-7482828173554917376-ycjU/?utm_source=share&utm_medium=member_desktop&rcm=ACoAADr4_oYBcuQmFTkaZrjX4750H068Dnnub5U&skipRedirect=true

[173] Y. Lee, R. S. Nair, Q. Zhang, K. Lee, O. Khattab, and C. Finn, "Meta-Harness: Post-Training Reliable Agent Systems via Harness Search," in *Second Workshop on Agents in the Wild: Safety, Security, and Beyond*, 2026. [Online]. Available: https://openreview.net/forum?id=2Tx03Dan7u

[174] M. Meier, "CrewHaus: A Meta-Harness Compiler for AI Agents," May 2026. [Online]. Available: https://crewhaus.ai/whitepaper/crewhaus-meta-harness-compiler.pdf

[175] R. Huang and S. Tao, "A human-centered automated machine learning agent with large language models for multimodal data management and analysis," *Frontiers in Artificial Intelligence*, vol. Volume 8 - 2025, 2025. doi: 10.3389/frai.2025.1680845. [Online]. Available: https://www.frontiersin.org/journals/artificial-intelligence/articles/10.3389/frai.2025.1680845

[176] Z. Liang, F. Wei, W. Xu, Y. Qian, L. Chen, and X. Wu, "I-MCTS: Enhancing Agentic AutoML via Introspective Monte Carlo Tree

Search," in *Findings of the Association for Computational Linguistics: EACL 2026*. Rabat, Morocco: Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-eacl.11 pp. 189–210. [Online]. Available: https://aclanthology.org/2026.findings-eacl.11

[177] S. Hoseini, M. Ibbels, M. Knoll, and C. Quix, "Towards enhancing data science agents with semantics," *Computer Science and Information Systems*, no. 00, pp. 78–78, 2025. doi: 10.2298/CSIS250320078H. [Online]. Available: https://doiserbia.nb.rs/Article.aspx?ID=1820-02142500078H

[178] S. Hoseini, A. Ali, H. Shaker, and C. Quix, "SEDAR: A Semantic Data Reservoir for Heterogeneous Datasets," in *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management*. Birmingham United Kingdom: ACM, Oct. 2023. doi: 10.1145/3583780.3614753. ISBN 979-8-4007-0124-5 pp. 5056–5060. [Online]. Available: https://dl.acm.org/doi/10.1145/3583780.3614753

[179] M. Fikardos, K. Lepenioti, A. Bousdekis, D. Apostolou, and G. Mentzas, "Generative AI for autonomous data analytics," *Intelligent Systems with Applications*, vol. 29, p. 200626, Mar. 2026. doi: 10.1016/j.iswa.2026.200626. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S2667305326000013

[180] W. Zhao, T. Chen, J. S. Yang, and L. Qiu, "AutoML-Pipeline: A RAG-Enhanced Code Generation Framework with Pre-validation for Cloud-Native Machine Learning Workflows," *IEEE Access*, pp. 1–1, 2026. doi: 10.1109/ACCESS.2026.3673923. [Online]. Available: https://ieeexplore.ieee.org/document/11433654/

[181] A. Jaffri, M. Hassanlou, T. Zhang, D. Seth, and Y. Bhatt, "Magic Quadrant for Data Science and Machine Learning Platforms," May 2025. [Online]. Available: https://www.gartner.com/en/documents/6533902

[182] Gartner, Inc., "Magic Quadrant Research Methodology," 2026. [Online]. Available: https://www.gartner.com/en/research/methodologies/magic-quadrants-research

[183] C. Guo, X. Liu, C. Xie, A. Zhou, Y. Zeng, Z. Lin, D. Song, and B. Li, "RedCode: Risky Code Execution and

Generation Benchmark for Code Agents," in *Advances in Neural Information Processing Systems*, A. Globerson, L. Mackey, D. Belgrave, A. Fan, U. Paquet, J. Tomczak, and C. Zhang, Eds., vol. 37. Curran Associates, Inc., 2024. doi: 10.52202/079017-3369 pp. 106 190–106 236. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2024/file/bfd082c452dffb450d5a5202b0419205-Paper-Datasets_and_Benchmarks_Track.pdf

[184] H. Trivedi, T. Khot, M. Hartmann, R. Manku, V. Dong, E. Li, S. Gupta, A. Sabharwal, and N. Balasubramanian, "AppWorld: A Controllable World of Apps and People for Benchmarking Interactive Coding Agents," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024. doi: 10.18653/v1/2024.acl-long.850 pp. 16 022–16 076. [Online]. Available: https://aclanthology.org/2024.acl-long.850/

[185] K. Zhang, J. Li, G. Li, X. Shi, and Z. Jin, "CodeAgent: Enhancing Code Generation with Tool-Integrated Agent Systems for Real-World Repo-level Coding Challenges," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024. doi: 10.18653/v1/2024.acl-long.737 pp. 13 643–13 658. [Online]. Available: https://aclanthology.org/2024.acl-long.737/

[186] Z. Ni, H. Wang, S. Zhang, S. Lu, Z. He, WangYou, Z. Tang, S. Hu, B. Li, C. Hu, B. Jiao, D. Jiang, Y. Du, and P. Lyu, "GitTaskBench: A Benchmark for Code Agents Solving Real-World Tasks Through Code Repository Leveraging," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 40, no. 38, pp. 32 564–32 572, Mar. 2026. doi: 10.1609/aaai.v40i38.40533. [Online]. Available: https://ojs.aaai.org/index.php/AAAI/article/view/40533

[187] K. Liu, Y. Pan, Y. Xiang, D. He, J. Li, Y. Du, and T. Gao, "ProjectEval: A Benchmark for Programming Agents Automated Evaluation on Project-Level Code Generation," in *Findings of the Association for Computational Linguistics: ACL 2025*, W. Che, J. Nabende, E. Shutova, and M. T. Pilehvar, Eds. Vienna, Austria: Association for

Computational Linguistics, Jul. 2025. doi: 10.18653/v1/2025.findings-acl.1036. ISBN 979-8-89176-256-5 pp. 20 205–20 221. [Online]. Available: https://aclanthology.org/2025.findings-acl.1036/

[188] M. Wang, X. Huang, J. Xie, S. Ma, J. Men, D. Liang, and Y. Cai, "From Model Diagram to Code: A Benchmark Dataset and Multi-Agent Framework," in *Proceedings of the 33rd ACM International Conference on Multimedia*. Dublin Ireland: ACM, Oct. 2025. doi: 10.1145/3746027.3755425. ISBN 979-8-4007-2035-2 pp. 1754–1763. [Online]. Available: https://dl.acm.org/doi/10.1145/3746027.3755425

[189] G. Yin, H. Bai, S. Ma, F. Nan, Y. Sun, Z. Xu, S. Ma, J. Lu, X. Kong, A. Zhang, D. A. Yap, Y. Zhang, K. Ahnert, V. Kamath, M. Berglund, D. Walsh, T. Gindele, J. Wiest, Z. Lai, X. S. Wang, J. Shan, M. Cao, R. Pang, and Z. Wang, "MMAU: A Holistic Benchmark of Agent Capabilities Across Diverse Domains," in *Findings of the Association for Computational Linguistics: NAACL 2025*, L. Chiruzzo, A. Ritter, and L. Wang, Eds. Albuquerque, New Mexico: Association for Computational Linguistics, Apr. 2025. doi: 10.18653/v1/2025.findings-naacl.267. ISBN 979-8-89176-195-7 pp. 4752–4780. [Online]. Available: https://aclanthology.org/2025.findings-naacl.267/

[190] J. J. Wu and F. H. Fard, "HumanEvalComm: Benchmarking the Communication Competence of Code Generation for LLMs and LLM Agents," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 7, pp. 1–42, Sep. 2025. doi: 10.1145/3715109. [Online]. Available: https://dl.acm.org/doi/10.1145/3715109

[191] A. Khatua, H. Zhu, P. Tran, A. Prabhudesai, F. Sadrieh, J. K. Lieberwirth, X. Yu, Y. Fu, M. J. Ryan, J. Pei, and D. Yang, "CooperBench: Benchmarking Cooperation in Coding Agents," in *Workshop on Multi-Agent Learning and Its Opportunities in the Era of Generative AI*, 2026. [Online]. Available: https://openreview.net/forum?id=AomNqiSwb1

[192] J. Gong, Y. Wu, L. Liang, Y. Wang, J. Chen, M. Liu, and Z. Zheng, "CoSQA+: Enhancing Code Search Evaluation With a Multi-Choice Benchmark and Test-Driven Agents," *IEEE Transactions on Software Engineering*, vol. 52, no. 1, pp. 206–220, 2026. doi:

10.1109/TSE.2025.3631886. [Online]. Available: https://www.computer.org/csdl/journal/ts/2026/01/11242152/2bBPHyUDHt6

[193] N. Butt, V. Chandrasekaran, N. Joshi, B. Nushi, and V. Balachandran, "BenchAgents: Automated Benchmark Creation with Agent Interaction," in *ICLR 2025 Workshop on Navigating and Addressing Data Problems for Foundation Models*, 2025. [Online]. Available: https://openreview.net/forum?id=Xh6S3X3enu

[194] Y. Zhang, Q. Jiang, X. XingyuHan, N. Chen, Y. Yang, and K. Ren, "Benchmarking Data Science Agents," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*. Bangkok, Thailand: Association for Computational Linguistics, 2024. doi: 10.18653/v1/2024.acl-long.308 pp. 5677–5700. [Online]. Available: https://aclanthology.org/2024.acl-long.308

[195] D. Zhang, S. Zhoubian, M. Cai, F. Li, L. Yang, W. Wang, T. Dong, Z. Hu, J. Tang, and Y. Yue, "DataSciBench: An LLM Agent Benchmark for Data Science," in *Findings of the Association for Computational Linguistics: ACL 2026*. San Diego, California, United States: Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-acl.181 pp. 3685–3728. [Online]. Available: https://aclanthology.org/2026.findings-acl.181

[196] M. Pietruszka, Ł. Borchmann, A. Jędrosz, and P. Morawiecki, "Can Models Help Us Create Better Models? Evaluating LLMs as Data Scientists," in *Findings of the Association for Computational Linguistics: EACL 2026*. Rabat, Morocco: Association for Computational Linguistics, 2026. doi: 10.18653/v1/2026.findings-eacl.308 pp. 5862–5886. [Online]. Available: https://aclanthology.org/2026.findings-eacl.308

[197] M. A. Merrill, A. G. Shaw, N. Carlini, B. Li, H. Raj, I. Bercovich, L. Shi, J. Y. Shin, T. Walshe, E. K. Buchanan, J. Shen, G. Ye, H. Lin, J. Poulos, M. Wang, J. Jitsev, M. Nezhurina, D. Lu, O. M. Mastromichalakis, Z. Xu, Z. Chen, Y. Liu, R. Zhang, L. L. Chen, A. Kashyap, J.-L. Uslu, J. Li, J. Wu, M. Yan, S. Bian, V. Sharma, K. Sun, S. Dillmann, A. Anand, A. Lanpouthakoun, B. Koopah, C. Hu, E. K. Guha, G. H. S. Dreiman, J. Zhu, K. Krauth, L. Zhong, N. Muennighoff, R. K. Amanfu, S. Tan, S. Pimpalgaonkar,

T. Aggarwal, X. Lin, X. Lan, X. Zhao, Y. Liang, Y. Wang, Z. Wang, C. Zhou, D. Heineman, H. Liu, H. Trivedi, J. Yang, J. Lin, M. Shetty, M. Yang, N. Omi, N. Raoof, S. Li, T. Y. Zhuo, W. Lin, Y. Dai, Y. Wang, W. Chai, S. Zhou, D. Wahdany, Z. She, J. Hu, Z. Dong, Y. Zhu, S. Cui, A. Saiyed, A. Kolbeinsson, C. M. Rytting, R. Marten, Y. Wang, A. Dimakis, A. Konwinski, and L. Schmidt, "Terminal-Bench: Benchmarking Agents on Hard, Realistic Tasks in Command Line Interfaces," in *The Fourteenth International Conference on Learning Representations*, 2026. [Online]. Available: https://openreview.net/forum?id=a7Qa4CcHak

[198] "Terminal-Bench," 2026. [Online]. Available: https://www.tbench.ai/benchmarks/terminal-bench-2?search=learning

[199] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Technical report, University of Toronto, Toronto, Ontario, Tech. Rep. 0, 2009, backup Publisher: University of Toronto. [Online]. Available: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf

[200] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional Architecture for Fast Feature Embedding," in *Proceedings of the 22nd ACM international conference on Multimedia*. Orlando Florida USA: ACM, Nov. 2014. doi: 10.1145/2647868.2654889. ISBN 978-1-4503-3063-3 pp. 675–678. [Online]. Available: https://dl.acm.org/doi/10.1145/2647868.2654889

[201] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, A. Desmaison, A. Köpf, E. Yang, Z. DeVito, M. Raison, A. Tejani, S. Chilamkurthy, B. Steiner, L. Fang, J. Bai, and S. Chintala, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," Dec. 2019, arXiv:1912.01703 [cs]. [Online]. Available: http://arxiv.org/abs/1912.01703

[202] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, and T. Mikolov, "FastText.zip: Compressing text classification models," *arXiv preprint arXiv:1612.03651*, 2016. [Online]. Available: https://arxiv.org/abs/1612.03651

[203] Y. LeCun and C. Cortes, "MNIST handwritten digit database," 2010. [Online]. Available: http://yann.lecun.com/exdb/mnist/

[204] M. Zechner, "MCP vs CLI: Benchmarking Tools for Coding Agents," Aug. 2025. [Online]. Available: https://mariozechner.at/posts/2025-08-15-mcp-vs-cli/

[205] ——, "What if you don't need MCP at all?" Nov. 2025, thread on Hacker News: https://news.ycombinator.com/item?id=45947444. [Online]. Available: https://mariozechner.at/posts/2025-11-02-what-if-you-dont-need-mcp/

[206] K. Yilmaz, "I Made MCP 94% Cheaper (And It Only Took One Command)," Feb. 2026, thread on Hacker News: https://news.ycombinator.com/item?id=47157398. [Online]. Available: https://kanyilmaz.me/2026/02/23/cli-vs-mcp.html

[207] R. Madabhushi, "MCP vs CLI: Benchmarking AI Agent Cost & Reliability," Mar. 2026. [Online]. Available: https://www.scalekit.com/blog/mcp-vs-cli-use

[208] J. Schmitt, "MCP vs. CLI for AI-native development," Mar. 2026. [Online]. Available: https://circleci.com/blog/mcp-vs-cli/

[209] C. Chen, "MCP is Dead; Long Live MCP!" Mar. 2026, thread on Hacker News: https://news.ycombinator.com/item?id=47380270. [Online]. Available: https://chrlschn.dev/blog/2026/03/mcp-is-dead-long-live-mcp/

[210] S. Kumak, "MCP vs CLI: Which Interface Do AI Agents Actually Prefer? - A Benchmark-Driven Comparison of Browser Automation Paradigms," Dec. 2025. [Online]. Available: https://gist.github.com/szymdzum/c3acad9ea58f2982548ef3a9b2cdccce

[211] S. Amzani, "Your MCP Server Is Eating Your Context Window. There's a Simpler Way," Mar. 2026, thread on Hacker News: https://news.ycombinator.com/item?id=47400261. [Online]. Available: https://www.apideck.com/blog/mcp-server-eating-context-window-cli-alternative

[212] Q. Huang, J. Vora, P. Liang, and J. Leskovec, "MLAgentBench: evaluating language agents on machine learning experimentation," in *Proceedings of the 41st International Conference on Machine Learning*, ser. ICML'24. JMLR.org, 2024. [Online]. Available: https://proceedings.mlr.press/v235/huang24y.html

[213] L. Edman and A. Fraser, "Mask and You Shall Receive: Optimizing Masked Language Modeling For Pretraining BabyLMs," in *Proceedings of the First BabyLM Workshop*, L. Charpentier, L. Choshen, R. Cotterell, M. O. Gul, M. Y. Hu, J. Liu, J. Jumelet, T. Linzen, A. Mueller, C. Ross, R. S. Shah, A. Warstadt, E. G. Wilcox, and A. Williams, Eds. Suzhou, China: Association for Computational Linguistics, Nov. 2025. doi: 10.18653/v1/2025.babylm-main.31 pp. 445–453. [Online]. Available: https://aclanthology.org/2025.babylm-main.31/

[214] D. Nathani, L. Madaan, N. Roberts, N. Bashlykov, A. Menon, V. Moens, M. Plekhanov, A. Budhiraja, D. Magka, V. Vorotilov, G. Chaurasia, D. Hupkes, R. S. Cabral, T. Shavrina, J. N. Foerster, Y. Bachrach, W. Y. Wang, and R. Raileanu, "MLGym: A New Framework and Benchmark for Advancing AI Research Agents," in *Second Conference on Language Modeling*, 2025. [Online]. Available: https://openreview.net/forum?id=ryTr83DxRq

[215] J. S. Chan, N. Chowdhury, O. Jaffe, J. Aung, D. Sherburn, E. Mays, G. Starace, K. Liu, L. Maksin, T. Patwardhan, A. Madry, and L. Weng, "MLE-bench: Evaluating Machine Learning Agents on Machine Learning Engineering," in *The Thirteenth International Conference on Learning Representations*, 2025. [Online]. Available: https://openreview.net/forum?id=6s5uXNWGIh

[216] Z. Jiang, D. Schmidt, D. Srikanth, D. Xu, I. Kaplan, D. Jacenko, and Y. Wu, "AIDE: AI-Driven Exploration in the Space of Code," Feb. 2025, arXiv:2502.13138 [cs]. [Online]. Available: http://arxiv.org/abs/2502.13138

[217] Y. Wang, Y. Zhang, Y. Wu, L. Lu, P. L. Nguyen, X. Wang, and C.-T. Nguyen, "MLAlgo-Bench: Can Machines Implement Machine Learning Algorithms?" in *Findings of the Association for Computational Linguistics: EMNLP 2025*, C. Christodoulopoulos, T. Chakraborty, C. Rose, and V. Peng, Eds. Suzhou, China: Association for Computational Linguistics, Nov. 2025. doi: 10.18653/v1/2025.findings-emnlp.772. ISBN 979-8-89176-335-7 pp. 14 298–14 329. [Online]. Available: https://aclanthology.org/2025.findings-emnlp.772/

[218] E. Liang, R. Marcus, S. Taneja, and Y. Zhang, "Are LLM agents good at join order optimization?" Apr. 2026. [Online]. Available: https://www.databricks.com/blog/are-llm-agents-good-join-order-optimization

[219] M. H. Erol, X. Hao, F. Bianchi, C. Greco, J. Tagliabue, and J. Zou, "Test-Time Optimization of Physical Query Plans with LLMs," Jun. 2026, arXiv:2602.10387 [cs.DB]. [Online]. Available: http://arxiv.org/abs/2602.10387

[220] S. Chaudhuri and U. Dayal, "An overview of data warehousing and OLAP technology," *ACM SIGMOD Record*, vol. 26, no. 1, pp. 65–74, Mar. 1997. doi: 10.1145/248603.248616. [Online]. Available: https://dl.acm.org/doi/10.1145/248603.248616

[221] N. R. Schneider, D. Ghilardi, G. Piccinini, and J. Tagliabue, ""Skill issues": data-centric optimization of lakehouse agents," May 2026, arXiv:2606.01185 [cs.AI]. [Online]. Available: http://arxiv.org/abs/2606.01185

[222] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2024. ISBN 978-3-662-69305-6 978-3-662-69306-3. [Online]. Available: https://link.springer.com/10.1007/978-3-662-69306-3

[223] F. Bacon, *Novum Organum: with other parts of the great instauration*, 3rd ed., P. Urbach, Ed. Chicago, Ill.: Open Court Publ, 1995. ISBN 978-0-8126-9245-7

[224] B. Wolff, "wolffbe/mlpab," Jul. 2026, original-date: 2026-07-14T13:20:39Z. [Online]. Available: https://github.com/wolffbe/mlpab

[225] J. Dowling, "jimdowling/mlfs-book," Mar. 2026, original-date: 2025-11-06T18:04:46Z. [Online]. Available: https://github.com/jimdowling/mlfs-book

[226] Open-Meteo.com, "Free Weather API," 2026. [Online]. Available: https://open-meteo.com/

[227] B. Wolff, "wolffbe/mlpab/tesbted/docs/tasks/README.md," Jul. 2026, original-date: 2026-07-14T13:20:39Z. [Online]. Available: https://github.com/wolffbe/mlpab/tree/main/testbed/docs/tasks

[228] Hopsworks AB, "Feature Store For ML," 2025. [Online]. Available: https://www.featurestore.org/

[229] Q. McNemar, "Note on the Sampling Error of the Difference Between Correlated Proportions or Percentages," *Psychometrika*, vol. 12, no. 2, pp. 153–157, Jun. 1947. doi: 10.1007/BF02295996. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0033312300045178/type/journal_article

[230] Student, "The Probable Error of a Mean," *Biometrika*, vol. 6, no. 1, p. 1, Mar. 1908. doi: 10.2307/2331554. [Online]. Available: https://www.jstor.org/stable/2331554?origin=crossref

[231] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 0th ed. Routledge, May 2013. ISBN 978-1-134-74270-7. [Online]. Available: https://www.taylorfrancis.com/books/9781134742707

[232] E. E. Cureton, "Rank-Biserial Correlation," *Psychometrika*, vol. 21, no. 3, pp. 287–290, Sep. 1956. doi: 10.1007/BF02289138. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0033312300047839/type/journal_article

[233] S. Holm, "A Simple Sequentially Rejective Multiple Test Procedure," *Scandinavian Journal of Statistics*, vol. 6, pp. 65–70, 1979. [Online]. Available: https://api.semanticscholar.org/CorpusID:122415379

[234] A. Agarwal and E. Goenawan, "Snowflake CoWork: The Personal Work Agent for Every Knowledge Worker," Jun. 2026. [Online]. Available: https://www.snowflake.com/en/blog/snowflake-cowork-personal-work-agent/

[235] "logicalclocks/hopsworks-api," May 2026, original-date: 2021-11-13T07:21:45Z. [Online]. Available: https://github.com/logicalclocks/hopsworks-api

[236] "logicalclocks/hopsworks-api/tree/main/python/hopsworks/cli," May 2026, original-date: 2021-11-13T07:21:45Z. [Online]. Available: https://github.com/logicalclocks/hopsworks-api/tree/main/python/hopsworks/cli

[237] "logicalclocks/hopsworks-api/tree/main/python/hopsworks/mcp," May 2026, original-date: 2021-11-13T07:21:45Z. [Online].

Available: https://github.com/logicalclocks/hopsworks-api/tree/main/python/hopsworks/mcp

[238] "logicalclocks/hopsworks-api/tree/main/skills," May 2026, original-date: 2021-11-13T07:21:45Z. [Online]. Available: https://github.com/logicalclocks/hopsworks-api/tree/main/skills

[239] L. Avstreikh, "Machine Learning at Terminal Velocity," Feb. 2026. [Online]. Available: https://www.youtube.com/watch?v=3zYXVTXqnQc

[240] Hopsworks AB, "Hopsworks 5.0 - Where AI Agents Go to Work - News," Jun. 2026. [Online]. Available: https://www.hopsworks.ai/news/hopsworks-5-0-where-ai-agents-go-to-work

[241] Databricks, "databricks/databricks-sdk-py," Jul. 2026. [Online]. Available: https://github.com/databricks/databricks-sdk-py

[242] ——, "databricks-feature-engineering," Jul. 2026.

[243] ——, "databricks/cli," Jul. 2026, original-date: 2022-05-13T12:12:36Z. [Online]. Available: https://github.com/databricks/cli

[244] ——, "databricks-solutions/ai-dev-kit," Jul. 2026. [Online]. Available: https://github.com/databricks-solutions/ai-dev-kit

[245] ——, "Use agents on Databricks," Jul. 2026. [Online]. Available: https://docs.databricks.com/aws/en/agents/agent-framework/build-agents

[246] P. Wendell, M. Zaharia, W. Hutchins, and G. Oshri, "Introducing Genie Code," Mar. 2026. [Online]. Available: https://www.databricks.com/blog/introducing-genie-code

[247] Databricks, "databricks/databricks-agent-skills," Jul. 2026, original-date: 2026-01-14T09:51:58Z. [Online]. Available: https://github.com/databricks/databricks-agent-skills

[248] S. Dwivedi, A. Attar, U. Unnikrishnan, and A. Yu, "Snowflake CoCo: The AI Coding Agent for the Modern Data Stack," Jun. 2026. [Online]. Available: https://www.snowflake.com/en/blog/snowflake-coco-ai-coding-agent-modern-data-stack/

Appendix A

Command Allowlist

This appendix reproduces the command allowlist enforced during every run, taken from the enforcement hook in the MLPlatformAgentBench (MLPAB) repository. A command is allowed if its program is the interface under test, a shell keyword or builtin, or a basic file and text utility. Every other program is denied before execution with an explanatory message, and every denial is recorded. The interface under test is the Python interpreter in the software development kit (SDK) condition, and in the command line interface (CLI) condition the platform's CLI binary restricted to its documented services together with its documented auxiliary binaries.

The allowed shell keywords and builtins are `if`, `then`, `elif`, `else`, `fi`, `for`, `while`, `until`, `do`, `done`, `case`, `esac`, `in`, `select`, `function`, `time`, `{`, `}`, `[[`, `]]`, `!`, `:`, `.`, `true`, `false`, `test`, `[`, `]`, `cd`, `pwd`, `echo`, `printf`, `export`, `set`, `unset`, `local`, `read`, `source`, `command`, `which`, `type`, `exit`, `return`, `pushd`, `popd`, `wait`, `shift`, `getopts`, `let`, `declare`, `alias`, and `umask`.

The allowed basic file and text utilities are `ls`, `cat`, `head`, `tail`, `tac`, `nl`, `wc`, `find`, `stat`, `file`, `du`, `df`, `tree`, `realpath`, `readlink`, `basename`, `dirname`, `cp`, `mv`, `mkdir`, `rmdir`, `rm`, `ln`, `touch`, `chmod`, `grep`, `egrep`, `fgrep`, `sed`, `awk`, `cut`, `sort`, `uniq`, `tr`, `diff`, `cmp`, `comm`, `tee`, `fold`, `column`, `rev`, `paste`, `join`, `xxd`, `od`, `hexdump`, `split`, `expand`, `unexpand`, `date`, `sleep`, `gzip`, `gunzip`, `zcat`, `bzip2`, `bunzip2`, `xz`, `unxz`, `tar`, `unzip`, `zip`, `shasum`, `md5sum`, `sha256sum`, and `cksum`.

These utilities move or inspect bytes, such as writing the local submission files and reading the task inputs, and perform no platform work outside the interface. The list deliberately excludes general purpose interpreters, such as

`node`, `ruby`, and `perl`, and network tools, such as `curl` and `wget`, which would allow an agent to bypass the interface under test. `sleep` is included so that an agent waiting on a remote job can pause between polls instead of spending a model turn on every poll, as a bounded foreground pause that performs no platform work, while backgrounding remains blocked.

Appendix B

Complete Market Assessment

Table B.1 reproduces the 23 scored coverage criteria of the 39 assessed platforms, generated directly from the assessment data in the accompanying repository, with each platform name and each mark linking to its documenting source in the digital version. The availability, headquarters, hosting, and skills facts are shown in Figures 4.2 and 4.3.

Table B.1: The 23 scored coverage criteria of the 39 assessed platforms, ordered by market coverage. F, T, I, and O denote the feature, training, inference, and operations stages, H denotes documented MCP server hosting, Py, Fs, and Db denote the PySpark, file-system, and dashboard capabilities available through ML platform agents, M denotes a documented meta-harness, S a published set of skills, and A documented agent deployments. A filled circle marks full support, an open circle partial support, and a dash no documented support. The trailing column O is not scored, and a dagger there records an other software agent, namely a software agent the vendor ships that is not an ML platform agent, such as one for analytics or office work, an assistant that suggests rather than executes, or a pipeline agent confined to another domain. In the digital version, each platform name links to the platform website and each mark links to the documenting source for its interaction class, where one is recorded.

	SDK				CLI				MCP					ML platform agents							MH	Sk	Ad	OA
Platform	F	T	I	O	F	T	I	O	F	T	I	O	H	F	T	I	O	Py	Fs	Db	M	S	A	O
Databricks	●	●	●	●	●	●	●	●	○	○	●	●	●	●	●	●	●	●	○	●	●	●	●	–
Amazon SageMaker	●	●	●	●	●	●	●	●	○	○	○	●	●	○	○	○	●	●	○	–	–	●	●	–
Google Gemini Enterprise Agent Platform (formerly Vertex AI)	●	●	○	●	●	●	○	●	–	○	○	○	●	–	–	–	–	–	–	–	–	●	●	†
Hopsworks	●	●	●	●	●	●	●	●	–	–	–	–	–	–	–	–	–	–	–	–	●	●	●	–

Table B.1: Scored coverage criteria (continued).

	SDK				CLI				MCP					ML platform agents							MH	Sk	Ad	OA
Platform	F	T	I	O	F	T	I	O	F	T	I	O	H	F	T	I	O	Py	Fs	Db	M	S	A	O
Microsoft Azure Machine Learning	●	●	○	●	●	●	○	●	–	–	○	○	–	–	–	–	–	–	–	–	–	●	●	†
Teradata	○	●	○	○	–	–	–	–	●	○	●	●	–	●	–	–	●	–	–	–	–	–	●	–
Iguazio	●	●	●	●	–	○	○	○	○	○	○	○	–	–	–	–	–	–	–	–	–	–	–	–
IBM Watson Studio	○	●	●	●	–	○	○	–	●	–	–	●	–	–	–	–	–	–	–	–	–	–	●	–
JFrog	●	●	●	○	○	●	●	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–	●	–
Snowflake	○	●	○	○	○	○	○	○	–	–	○	–	–	–	–	–	–	–	–	–	●	–	●	†
FeatureByte	●	●	●	–	–	–	–	–	–	–	–	–	–	●	●	●	●	–	–	–	–	–	–	–
SAS Machine Learning	–	●	●	○	○	–	–	–	●	●	○	○	–	–	–	–	–	–	–	–	–	–	●	†
DataRobot	–	●	●	●	–	○	○	–	–	–	–	–	●	–	–	–	–	–	–	–	–	●	●	–
H2O.ai	–	●	●	○	–	○	○	–	○	–	●	○	–	–	–	–	–	–	–	–	–	–	●	–
Domino Data Lab	–	●	●	●	○	○	○	○	–	○	–	–	–	–	–	–	–	–	–	–	–	●	–	–
Dataiku	–	●	●	●	○	●	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–	–	–	–
Cloudera	–	–	–	–	–	○	○	○	○	●	○	●	–	–	–	–	–	–	–	–	–	–	●	†
Abacus	●	●	●	●	–	–	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–	–	–	†
Modal	–	○	○	○	–	○	○	○	–	–	–	–	●	–	–	–	–	–	–	–	–	–	●	–
Bauplan	○	–	–	○	○	–	–	○	●	–	○	○	–	–	–	–	–	–	–	–	–	–	–	†
Dremio	–	–	–	○	–	–	–	○	○	–	○	●	–	–	–	–	–	–	–	–	–	●	–	†
Astronomer	–	○	–	○	–	–	–	○	○	○	○	●	–	–	–	–	–	–	–	–	–	–	–	†
Alteryx	–	○	○	○	○	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Alibaba Cloud Platform for AI	–	–	–	–	–	○	○	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–	●	–
Chronon	●	○	○	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
OpenMLDB	●	–	–	–	●	–	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
MathWorks	–	–	–	–	–	–	–	–	○	○	○	○	–	–	–	–	–	–	–	–	–	–	–	†
KNIME	–	–	–	–	–	–	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–	–	●	–
Featureform	–	–	–	–	○	–	○	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Chalk	●	–	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
mloda.ai	○	○	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
TIBCO Software	–	–	–	–	–	–	–	–	–	–	–	–	●	–	–	–	–	–	–	–	–	–	–	†
Aible	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	●	†
Feathr	●	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Altair	–	–	–	–	–	–	–	○	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Apple Overton	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Facebook FBLearner	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Spotify Ray	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–
Uber Michelangelo	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–	–

TRITA – XXX-EX 2025:0000
Stockholm, Sweden 2026

www.kth.se

€€€€ For DIVA €€€€

{
"Author1": { "Last name": "Wolff",
"First name": "Benedict",
"Local User Id": "bjpwolff",
"E-mail": "bjpwolff@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
}
},
"Cycle": "2",
"Course code": "DA258X",
"Credits": "30.0",
"Degree1": {"Educational program": "Master's Programme, ICT Innovation, 120 credits"
,"programcode": "TIVNM"
,"Degree": "Degree of Master of Science"
,"subjectArea": "Computer Science and Engineering"
},
"Title": {
"Main title": "MLPlatformAgentBench",
"Subtitle": "Can coding agents optimize machine learning platform interfaces?",
"Language": "eng" },
"Alternative title": {
"Main title": "MLPlatformAgentBench",
"Subtitle": "Kan kodningsagenter optimera gränssnitt för maskininlärningsplattformar?",
"Language": "swe"
},
"Supervisor1": { "Last name": "Veresov",
"First name": "Aleksey",
"E-mail": "aleksei.veresov@cispa.de",
"Other organisation": "CISPA Helmholtz Center for Information Security, Doctoral Researcher"
},
"Supervisor2": { "Last name": "Dowling",
"First name": "Jim",
"E-mail": "jim@hopsworks.ai",
"Other organisation": "Hopsworks AB, CEO"
},
"Examiner1": { "Last name": "Payberah",
"First name": "Amir Hossein",
"Local User Id": "payberah",
"E-mail": "payberah@kth.se",
"organisation": {"L1": "School of Electrical Engineering and Computer Science",
"L2": "SCS" }
},
"Cooperation": { "Partner_name": "Hopsworks AB"},
"National Subject Categories": "10201, 10202",
"SDGs": "8, 9",
"Other information": {"Year": "2026", "Number of pages": "xviii,186"},
"Copyrightleft": "copyright",
"Series": { "Title of series": "TRITA – XXX-EX" , "No. in series": "2025:0000" },
"Opponents": { "Name": "Emil Lidbom"},
"Presentation": { "Date": "2026-08-17 13:00"
,"Language":"eng"
,"Room": "via Zoom https://kth-se.zoom.us/my/payberah"
,"Address": "Lindstedtsvägen 30, 114 28 Stockholm"
,"City": "Stockholm" },
"Number of lang instances": "2",

"Abstract[eng]": €€€€
Coding agents are operating machine learning platforms, yet no systematic study has established which platforms they can operate, and at what cost. This thesis maps and measures that landscape.
A market assessment scores 39 machine learning platforms by whether an agent can manage feature, training, and inference pipelines through a Python software development kit, a command line interface, or a Model Context Protocol server. It also examines platform agents and platform meta-harnesses. Five platforms provide complete coverage through at least one of the first two interfaces, and three through both. One of those three is European. None provides full Model Context Protocol coverage.
MLPlatformAgentBench evaluates the interfaces on 22 tasks from feature engineering through operations, restricting each agent to one interface and basic shell tools. Across Hopsworks and Databricks, two American and two European models, two interfaces, and, for Claude, two skill conditions, coding agent and model choice made the largest performance difference. Claude Opus in Claude Code solved 98% of runs, compared with 36% for Mistral Large in Mistral Vibe. The interface changed how work was done, not whether it succeeded. When the command line interface and software development kit were similarly mature, progressive disclosure offered no measurable advantage. Also, skills had no detectable effect on Claude Code's accuracy. Operating platforms differed detectably in accuracy only for Mistral Large in Mistral Vibe, which was more accurate on Hopsworks. The Hopsworks configuration also used fewer turns, less local time, and lower Claude model invocation cost, but the design does not isolate platform from execution period and coding agent version. The mean Claude run cost 1.06 against 2.32 US dollars. Claude Code then produced six optimized command line interface variants and one optimized set of skills for each optimization. None beat its corresponding baseline. This null result supports treating agent-based interface optimizations as hypotheses requiring paired measurement.

"Abstract[swe]": €€€€

Kodningsagenter använder maskininlärningsplattformar, men ingen systematisk studie har fastställt vilka plattformar de kan använda och till vilken kostnad. Denna avhandling kartlägger och mäter detta landskap.
En marknadsbedömning poängsätter 39 maskininlärningsplattformar utifrån om en agent kan hantera funktions-, tränings- och inferenspipelines via ett Python-programvaruutvecklingskit, ett kommandoradsgränssnitt eller en Model Context Protocol-server. Den undersöker också

plattformsagenter och plattformsmeta-harnesses. Fem plattformar ger fullständig täckning genom minst ett av de två första gränssnitten, och tre genom båda. Ett av dessa tre är europeiskt. Ingen ger fullständig Model Context Protocol-täckning.
MLPlatformAgentBench utvärderar gränssnitten på 22 uppgifter från funktionsutveckling till drift, och begränsar varje agent till ett gränssnitt och grundläggande skalverktyg. För Hopsworks och Databricks gjorde två amerikanska och två europeiska modeller, två gränssnitt och, för Claude, två färdighetsvillkor, kodningsagent och modellval den största prestandaskillnaden. Claude Opus i Claude Code löste 98% av körningarna, jämfört med 36% för Mistral Large i Mistral Vibe. Gränssnittet förändrade hur arbetet utfördes, inte huruvida det lyckades. När kommandoradsgränssnittet och programvaruutvecklingspaketet var lika mogna, erbjöd progressiv avslöjande ingen mätbar fördel. Färdigheter hade inte heller någon märkbar effekt på Claude Codes noggrannhet. Operativplattformar skilde sig märkbart åt i noggrannhet endast för Mistral Large i Mistral Vibe, vilket var mer exakt på Hopsworks. Hopsworks-konfigurationen använde också färre varv, mindre lokal tid och lägre kostnad för att anropa Claude-modellen, men designen isolerar inte plattformen från exekveringsperiod och kodningsagentversion. Den genomsnittliga Claude-körningen kostade 1,06 jämfört med 2,32 amerikanska dollar. Claude Code producerade sedan sex optimerade kommandoradsgränssnittsvarianter och en optimerad uppsättning färdigheter för varje optimering. Ingen slog motsvarande baslinje. Detta nollresultat stöder behandling av agentbaserade gränssnittsoptimeringar som hypoteser som kräver parade mätningar. €€€€,
"Keywords[eng]": €€€€
Machine Learning Platforms, Coding Agents, Large Language Models, Benchmark, Command Line Interface, Software Development Kit, Agent Skills, Sovereign Artificial Intelligence €€€€,
€€€€,
"Keywords[swe]": €€€€
Maskininlärningsplattformar, Kodagenter, Stora språkmodeller, Riktmärke, Kommandoradsgränssnitt, Programmeringsgränssnitt, Agentfärdigheter, Suverän artificiell intelligens €€€€,
}

acronyms.tex

```
%%% Local Variables:
%%% mode: latex
%%% TeX-master: t
%%% End:
% The following command is used with glossaries-extra
\setabbreviationstyle[acronym]{long-short}
% The form of the entries in this file is \newacronym{label}{acronym}{phrase}
%                                      or \newacronym[options]{label}{acronym}{phrase}
% see "User Manual for glossaries.sty" for the  details about the options, one example is shown below
% note the specification of the long form plural in the line below
% \newacronym[longplural={Debugging Information Entities}]{DIE}{DIE}{Debugging Information Entity}
%
% The following example also uses options
% \newacronym[shortplural={OSes}, firstplural={operating systems (OSes)}]{OS}{OS}{operating system}

% \newacronym{IQL}{IQL}{Independent Q-Learning}

% example of putting in a trademark on first expansion
% \newacronym[first={NVIDIA OpenSHMEM Library (NVSHMEM\texttrademark)}]{NVSHMEM}{NVSHMEM}{NVIDIA OpenSHMEM Li

% A
\newacronym{AAAI}{AAAI}{Association for the Advancement of Artificial Intelligence}
\newacronym{AI}{AI}{artificial intelligence}
\newacronym{AIDE}{AIDE}{Artificial Intelligence-Driven Exploration in the Space of Code}
\newacronym{ALF}{ALF}{Action Learning From}
\newacronym{ALFRED}{ALFRED}{Action Learning From Realistic Environments and Directives}
\newacronym{Amazon S3}{Amazon S3}{Amazon Simple Storage Service}
\newacronym{AMQP}{AMQP}{Advanced Message Queuing Protocol}
\newacronym{API}{API}{application programming interface}
\newacronym{AppWorld}{AppWorld}{Application World}
\newacronym{AQA}{AQA}{analysis questions and answers}
\newacronym{AST}{AST}{abstract syntax tree}
\newacronym{AUC}{AUC}{area under curve}
\newacronym{AutoML}{AutoML}{automated machine learning}
\newacronym{AWS}{AWS}{Amazon Web Services}

% B
\newacronym{B}{B}{billion}
\newacronym{BabyLM}{BabyLM}{Baby Language Model}
\newacronym{BASH}{BASH}{Bourne Again SHell}
\newacronym{BDI}{BDI}{belief-desire-intention}
\newacronym{Bench}{bench}{benchmark}
\newacronym{BERT}{BERT}{Bidirectional Encoder Representations from Transformers}
\newacronym{BFCL}{BFCL}{Berkeley Function Calling Leaderboard}
\newacronym{BI}{BI}{business intelligence}
\newacronym{BM25}{BM25}{Okapi Best Matching 25}
\newacronym{BSD}{BSD}{Berkeley Software Distribution}

% C
\newacronym{CASSIA}{CASSIA}{Collective Agent System for Single-Cell Interpretable Annotation}
\newacronym{CI/CD}{CI/CD}{continuous integration and continuous delivery}
\newacronym{CIFAR}{CIFAR}{Canadian Institute for Advanced Research}
\newacronym{CLAI}{CLAI}{Command Line Artificial Intelligence}
\newacronym{CLI}{CLI}{command line interface}
\newacronym{CNN}{CNN}{convolutional neural network}
\newacronym{CoSQA}{CoSQA}{Code Semantic Question Answering}
\newacronym{CSV}{CSV}{comma-separated values}

% D
\newacronym{DA}{DA}{Data Analysis}
\newacronym{DAI}{DAI}{distributed artificial intelligence}
\newacronym{DB}{DB}{database}
\newacronym{DBN}{DBN}{deep belief network}
\newacronym{DataSciBench}{DataSciBench}{Data Science Benchmark}
\newacronym{DBPlanBench}{DBPlanBench}{Database Plan Benchmark}
\newacronym{DeepEval}{DeepEval}{Deep Evaluation}
\newacronym{dev}{dev}{development}
\newacronym{DevOps}{DevOps}{development and operations}
\newacronym{DevTool}{DevTool}{development tool}
\newacronym{DGM}{DGM}{deep generative model}
\newacronym{DL}{DL}{deep learning}

% E
\newacronym{eng}{eng}{engineering}
\newacronym{Eval}{eval}{evaluation}
\newacronym{EU}{EU}{European Union}
```

```
% F
\newacronym{FB}{FB}{Facebook}
\newacronym{feat}{feat}{feature}
\newacronym{FeatEng}{FeatEng}{Feature Engineering}
\newacronym{FS}{FS}{file system}
\newacronym{FTI}{FTI}{feature, training and inference}

% G
\newacronym{GAN}{GAN}{generative adversarial network}
\newacronym{GB}{GB}{gigabyte}
\newacronym{GCP}{GCP}{Google Cloud Platform}
\newacronym{GDPR}{GDPR}{General Data Protection Regulation}
\newacronym{GenAI}{GenAI}{generative artificial intelligence}
\newacronym{GEPA}{GEPA}{Genetic-Pareto}
\newacronym{GPT}{GPT}{generative pre-trained transformer}
\newacronym{GPU}{GPU}{graphics processing unit}

% H
\newacronym{H2O}{H2O}{Dihydrogen Monoxide}
\newacronym{HDFS}{HDFS}{Hadoop Distributed File System}
\newacronym{HN}{HN}{Hacker News}
\newacronym{HopsFS}{HopsFS}{Hopsworks File System}
\newacronym{HTTP}{HTTP}{Hypertext Transfer Protocol}
\newacronym{HumanEvalComm}{HumanEvalComm}{Human Evaluation Communication}

% I
\newacronym{I-MCTS}{I-MCTS}{Introspective Monte Carlo Tree Search}
\newacronym{IBM}{IBM}{International Business Machines}
\newacronym{IDE}{IDE}{integrated development environment}
\newacronym{IEEE}{IEEE}{Institute of Electrical and Electronics Engineers}
\newacronym{IPYNB}{IPYNB}{Interactive Python Notebook}

% J
\newacronym{JSON}{JSON}{JavaScript Object Notation}

% K
\newacronym{KIF}{KIF}{Knowledge Interchange Format}
\newacronym{KNIME}{KNIME}{Konstanz Information Miner}
\newacronym{KQML}{KQML}{Knowledge Query and Manipulation Language}

% L
\newacronym{LLaMA}{LLaMA}{Large Language Model Meta Artificial Intelligence}
\newacronym{LLM}{LLM}{large language model}
\newacronym{LLM2AutoML}{LLM2\-Auto\-ML}{Large Language Model to Automated Machine Learning}

% M
\newacronym{MCP}{MCP}{Model Context Protocol}
\newacronym{MD}{MD}{Markdown}
\newacronym{MDC}{MDC}{Model Diagram Code}
\newacronym{MIT}{MIT}{Massachusetts Institute of Technology}
\newacronym{ML}{ML}{machine learning}
\newacronym{MLE}{MLE}{machine learning engineering}
\newacronym{MLP}{MLP}{multi-layer perceptron}
\newacronym{MLPAB}{MLPAB}{MLPlatformAgentBench}
\newacronym{MLOps}{MLOps}{machine learning operations}
\newacronym{MMAU}{MMAU}{Massive Multitask Agent Understanding}
\newacronym{MNIST}{MNIST}{Modified National Institute of Standards and Technology}
\newacronym{MAS}{MAS}{multi-agent system}

% N
\newacronym{NAS}{NAS}{neural architecture search}
\newacronym{net}{net}{network}
\newacronym{NeurIPS}{NeurIPS}{Conference on Neural Information Processing Systems}
\newacronym{NLC2CMD}{NLC2\-CMD}{Natural Language Command to Command}
\newacronym{NLP}{NLP}{natural language processing}
\newacronym{NumPy}{NumPy}{Numerical Python}

% O
\newacronym{OAuth}{OAuth}{Open Authentication}
\newacronym{OLAP}{OLAP}{Online Analytical Processing}
\newacronym{OpenAI}{OpenAI}{Open Artificial Intelligence}
\newacronym{OpenMLDB}{OpenMLDB}{Open Machine Learning Database}
\newacronym{OS}{OS}{operating system}

% P
\newacronym{POSIX}{POSIX}{Portable Operating System Interface}
\newacronym{PM2.5}{PM2.5}{Particulate Matter with an Aerodynamic Diameter of 2.5\,\textmu m or less}

% R
```

```
\newacronym{RAG}{RAG}{retrieval-augmented generation}
\newacronym{RBM}{RBM}{Restricted Boltzmann Machine}
\newacronym{ReLU}{ReLU}{rectified linear unit}
\newacronym{REPL}{REPL}{read-eval-print loop}
\newacronym{REST}{REST}{Representational State Transfer}
\newacronym{RL}{RL}{reinforcement learning}
\newacronym{RLHF}{RLHF}{reinforcement learning from human feedback}
\newacronym{RMSE}{RMSE}{root mean square error}
\newacronym{RNA}{RNA}{ribonucleic acid}
\newacronym{ROC}{ROC}{receiver operating characteristic}
\newacronym{RQ}{RQ}{research question}

% S
\newacronym{S}{S}{source}
\newacronym{SaaS}{SaaS}{software as a service}
\newacronym{SAP}{SAP}{System Applications and Products in Data Processing}
\newacronym{SAS}{SAS}{Statistical Analysis System}
\newacronym{scikit}{scikit}{Scientific Python Toolkit}
\newacronym{SciSciGPT}{SciSciGPT}{Science Science Generative Pre-trained Transformer}
\newacronym{SDK}{SDK}{software development kit}
\newacronym{SECOM}{SECOM}{Semiconductor Manufacturing}
\newacronym{SEDAR}{SEDAR}{Semantic Data Reservoir for Heterogeneous Datasets}
\newacronym{SGD}{SGD}{stochastic gradient descent}
\newacronym{SOTA}{SOTA}{state of the art}
\newacronym{SQL}{SQL}{structured query language}
\newacronym{SS}{SS}{search string}
\newacronym{SWE}{SWE}{software engineering}

% T
\newacronym{TIBCO}{TIBCO}{The Information Bus Company}
\newacronym{TPC-DS}{TPC-DS}{Transaction Processing Performance Council decision support benchmark}
\newacronym{TPC-H}{TPC-H}{Transaction Processing Performance Council ad hoc decision support benchmark}
\newacronym{TUI}{TUI}{text-based user interface}
\newacronym{TTY}{TTY}{teletypewriter}

% U
\newacronym{USD}{USD}{United States Dollar}

% V
\newacronym{VAE}{VAE}{variational autoencoder}

% X
\newacronym{XGBoost}{XGBoost}{eXtreme Gradient Boosting}

% Y
\newacronym{YAML}{YAML}{YAML Ain't Markup Language}
```