# Software 3.0

Saeed Siddik
Assistant Professor
IIT University of Dhaka

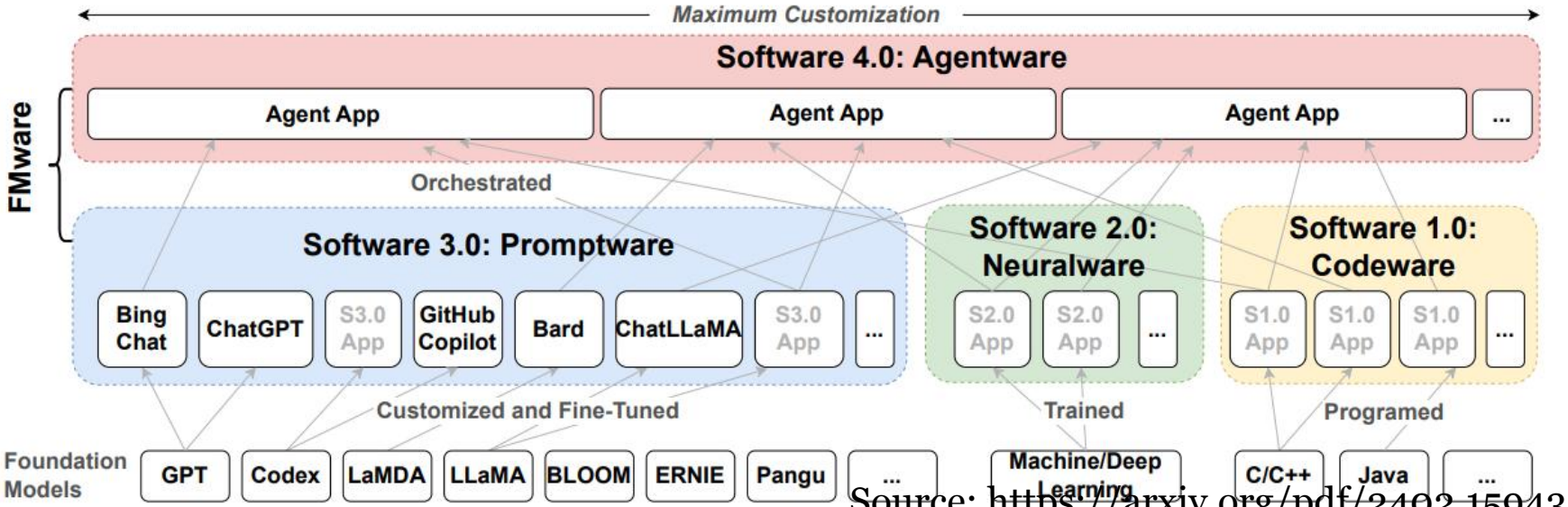
# Evolution on Software Engineering

- SE 1.0 – The Code-Centric Era
  - Classical Software Engineering
- SE 2.0 – The Data, Service and Collaboration Era
  - DevOps and ML driven training-prediction
- SE 3.0 – The AI-Driven and Autonomous Era
  - LLMs and AIware in software automation

# SE 3.0 – The AI-Driven and Autonomous Era (Promptware)

- Core Idea: Software is customized and fine-tuned through foundation models (FMs) such as GPT, using prompts instead of training from scratch.
- Process: Developers and users interact with pre-trained AI-models through prompt engineering and fine-tuning.
- Examples: ChatGPT, GitHub Copilot, Bard, and LLaMA.
- Characteristics:
  - Built on top of massive pre-trained models (foundation models).
  - Bridges AI with software engineering via prompt-based design.

# Agentware and its relation to prior software generations.

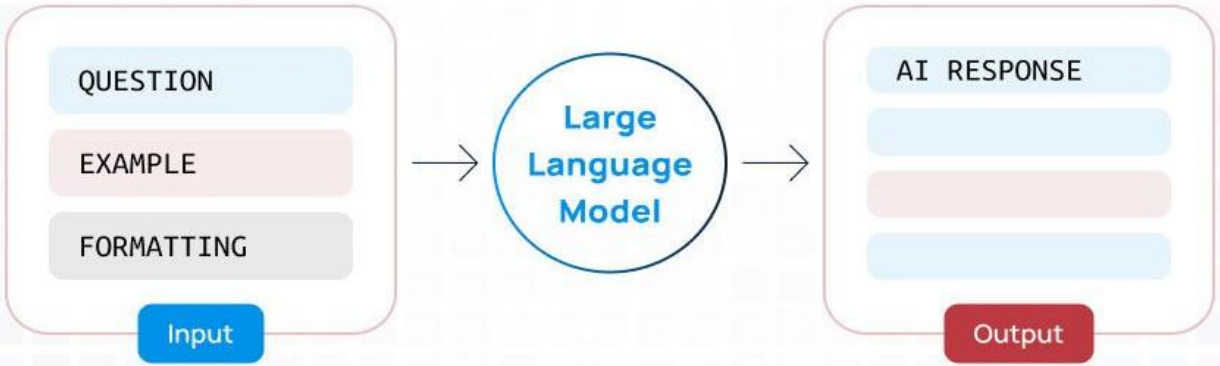



Source: https://arxiv.org/pdf/2402.15943

# Developing AIware systems

- Prompt Engineering
- Fine-tuning
- Retrieval Augmented Generation (RAG)

# Prompt Engineering



# Prompt Design Strategies

- Instructional Prompts: direct commands (e.g., “Generate test cases for this function”).
- Contextual Prompts: provide background, data, or examples.
- Role-based Prompts: define persona (“You are a DevOps assistant…”).
- Chain-of-Thought Prompts: encourage stepwise reasoning.
- System Prompts: configure persistent behavior of AI agents.

# Example of Instructional Prompt: direct command

- **Prompt:** "`Generate 5 unit test cases in Python (using pytest) for the function 'compute_mean(values: List[float])' covering normal, empty, and error cases; include expected inputs and outputs.`"

- Expected output: A list of pytest test functions with inputs, expected outputs, and one test that checks raised exception for invalid input.

# Example of Contextual Prompt: provide background/data/examples

- Prompt: "`Given the following bug report and recent commit diff (paste below), summarize the root cause and propose a code patch sketch. Bug report: 'API returns 500 when input size > 10MB.' Commit diff: <paste>.`"

- Expected output: A concise root-cause analysis referencing the diff, and a short patch outline (file, function, pseudo-code) that addresses buffering/streaming or validation.

# Example of Role-based Prompt: define a persona / responsibilities

- Prompt: "`You are a Senior DevOps Engineer. Review this CI pipeline (YAML below) and list three improvements to reduce deployment failures and one rollback strategy.`"

- Expected output: Three actionable CI/CD improvements (e.g., add integration tests, implement canary releases, tighten resource limits) and a clear rollback procedure.

# Example of Chain-of-Thought Prompt: encourage stepwise reasoning

- Prompt: "`Explain step-by-step how you would find the root cause of intermittent test failures in a microservice: list hypotheses, tests to run for each hypothesis, expected observations, and next actions.`"

- Expected output: A numbered reasoning chain: hypothesis A → test A1 → expected outcome → next step, then hypothesis B → … useful for teaching debugging workflows.

# Example of System Prompt: configure persistent agent behavior

- Prompt: "`Explain step-by-step how you would find the root cause of intermittent test failures in a microservice: list hypotheses, tests to run for each hypothesis, expected observations, and next actions.`"

- Expected output: A numbered reasoning chain: hypothesis A → test A1 → expected outcome → next step, then hypothesis B → … useful for teaching debugging workflows.

# Example of System Prompt: configure persistent agent behavior

- Prompt: "`System: You are 'CodeQA', an assistant that always returns answers in three parts—(1) brief summary, (2) detailed explanation with references to code lines, (3) suggested test cases. Never provide legal or private data.`"

- Expected output: All subsequent responses follow the three-part structure, producing consistent, policy-compliant replies suitable for integration into an AIware

# Best Practices in Prompt Engineering

- Be specific: define output format and success criteria.
- Add context: mention task scope, tools, or dataset.
- Use examples: to guide tone, structure, or code style.
- Avoid ambiguity: test prompts iteratively.
- Evaluate responses: refine prompts like you debug software.

# Prompt Engineering Best Practices

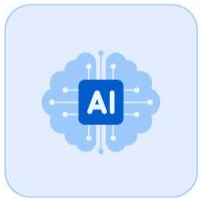
Specify an Audience

Be Clear and Specific

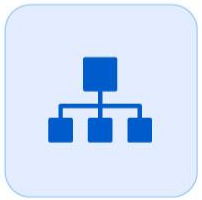
Set a Persona

Comprehend the Task at Hand

Remove Ambiguity




Tell AI What Not to Do



Break Down Complex Tasks


Structure Prompts by Priority

Specify the Output Format

# Prompt Engineering in SDLC

- In AI-native projects, prompts evolve like software artifacts.
- Managed through version control and collaborative prompt libraries.
- Used in:
  - Requirement elicitation (e.g., “Describe system constraints…”)
  - Testing & QA (e.g., “Find logical errors in this code…”)
  - Deployment (defining system prompt for agents).
- Use tools like GitHub or prompt management systems (e.g., PromptHub, LangSmith).
- Prompts become reusable project assets, like source code or test cases.

# Testing and Refining Prompts for Quality Output

- Evaluate prompt performance using:
  - Accuracy: Does output meet task requirements?
  - Consistency: Are results reproducible?
  - Relevance: Does it align with context and constraints?
- Refinement process = Prompt–Response–Review–Adjust cycle.
- Use feedback loops or automatic scoring tools (BLEU, ROUGE for text).

# BLEU and ROUGE score to assess prompt and response quality

- BLEU (Bilingual Evaluation Understudy):
  - Measures precision : how much generated text overlaps with reference text.
  - Common in machine translation and code generation tasks.
  - Higher BLEU → closer match to reference output.
- ROUGE (Recall-Oriented Understudy for Gisting Evaluation):
  - Measures recall : how much of the reference text is captured by generated output.
  - Used for summarization, report generation, and explanation tasks.
  - Higher ROUGE → better content coverage.

# Ethical and Responsible Prompting

- Prompts can unintentionally induce bias, hallucination, or data leakage.
- Always validate model outputs against ethical standards and data governance policies.
- Avoid prompts that:
  - Encourage harmful or biased outputs.
  - Reveal private or copyrighted data.
- Part of responsible AI project management.

# Developing AIware systems

- Prompt Engineering
- Fine-tuning
- Retrieval Augmented Generation (RAG)

# Fine tuning

- adapting a pre-trained AI-Model to perform better on a specific task or with a domain-specific dataset.
- The goal is to adapt the model's general knowledge to a specialized task, domain, or style.

# Why Fine-Tune? The PM's View

### Training from Scratch

Requires **massive** datasets

Extremely high compute cost

Very high risk of failure

Impractical for 99% of software projects

### Fine-Tuning

Leverages existing knowledge

Requires a smaller, curated dataset

(100s-1000s of examples)

Drastically lower cost and time

Faster path to a specialized model

# When NOT to fine-tune

- If non-compliant with privacy/regulatory constraints and model contains sensitive pretraining data you cannot modify.
- When task requires radically different architecture/inductive biases.
- When you must guarantee zero change from pretrained weights (use inference-time adapters instead).
- When compute budget or latency demands prohibit larger models like considering smaller models or distilled variants.

# Types of Fine-Tuning (Part 1)

## 1. Full Fine-Tuning

Updates **all** weights of the pre-trained model. It's the most thorough method but is computationally expensive and creates a full-size copy of the model.

## 2. PEFT

**P**arameter-**E**fficient **F**ine-**T**uning. Updates only a **small subset** of parameters (or adds new ones). It's much faster, cheaper, and creates tiny "adapter" files (e.g., 50MB vs 15GB).

# Types of Fine-Tuning (Part 2): PEFT

Popular Parameter-Efficient Fine-Tuning Methods

- **LoRA (Low-Rank Adaptation):** Injects small, trainable "adapter" matrices into the model. The original model weights are frozen. This is the most popular and effective method.

- **QLoRA (Quantized LoRA):** An optimization of LoRA. Uses quantization to reduce the model's memory footprint, allowing fine-tuning of huge models on smaller, cheaper GPUs.

# Project workflow of Fine-Tuning Process

### 1. Task Definition

Classify taregt question and requirement

### 2. Data Prep

Collect & label 1,000 example with the correct label.

### 3. Model Selection

Choose a base model, e.g., 'BERT-base' or 'Llama-3 8B'.

### 4. Tuning

Run the training data. Feed the 1,000 examples using LoRA.

### 5. Evaluation

Test the new model on unseen data. Check its accuracy.

# Example: Task and data

- **Task**: sentiment classification for customer reviews (3 classes: positive, neutral, negative).
- **Pretrained model**: BERT-base (or a Transformer encoder).
- **Dataset**: 10k labeled reviews, 80/10/10 train/val/test.
- **Evaluation metric**: accuracy + F1 (macro).

# Project management for fine-tuned AI systems

- **Phases**: discovery → data collection → modeling → integration → testing → deployment → monitoring → maintenance.
- **Roles**: Project Manager, Software Engineer, ML Engineer, Data Engineer, Annotator, QA, DevOps, UX.
- **Deliverables**: dataset spec, baseline, model artifacts, evaluation report, monitoring plan, runbooks.

# Developing AIware systems

- Prompt Engineering
- Fine-tuning
- Retrieval Augmented Generation (RAG)

# What is RAG?

- Retrieval-Augmented Generation (RAG) is an AI architecture that combines an information retrieval system with a language model
- Responses are generated using both the model's knowledge and externally retrieved, up-to-date data.
- RAG enables AI-enabled software products to be more accurate, maintainable, and adaptable by separating knowledge management (indexing and retrieval) from language understanding and response creation (generation).

# How RAG system works?

# Indexing stage

- knowledge sources such as documents, books, databases, or policies are collected, cleaned, split into smaller chunks,
- converted into vector embeddings,
- stored in a searchable index (usually a vector database),
- prepares the knowledge for efficient access.

# Retrieval stage

- when a user asks a question, the system converts the query into an embedding
- searches the index to find the most relevant pieces of information,
- often filtering or re-ranking them to improve relevance and accuracy.

# Generation stage

- the retrieved content is injected into the prompt of a language model,
- then generates a response that is grounded in the retrieved evidence,
- reducing hallucination and increasing trustworthiness.

# RAQ vs Fine Tuning

| Aspect | RAG | Fine-Tuning |
|---|---|---|
| Basic Idea | Retrieves external knowledge at query time and uses it during generation | Embeds knowledge directly into the model's weights |
| Knowledge Update | Easy to update by change/add documents | Requires retraining / re-fine-tuning the model |
| Data Freshness | Can use real-time or frequently updated data | Limited to data available during training |
| Hallucination Risk | Lower, as answers are grounded in sources | Higher, since model relies on internal memory |
| Explainability | High, retrieved documents can be shown as evidence | Low, model decisions are not easily traceable |
| Scalability | Scales well by expanding the knowledge | Poor scalability as data size grows |
| Latency | Slightly higher due to retrieval step | Lower at inference time |
| Best Use Cases | Enterprise search, QA systems, policy assistants | Style adaptation, domain language learning |
| Software Engineering Fit | Modular and aligns with modern software architecture | Tightly coupled to the model |
| AIWare Projects | Yes, for most production systems | Only for narrow, stable domains |

# End of Software 3.0