Module 2: Prompt Engineering Fundamentals

CS5013 – Programming with AI

V. Krishna Nandivada

Department of Computer Science and Engineering
Indian Institute of Technology Madras

Module 2 at a glance (12 lecture hours)

1. Anatomy of a prompt
2. Specificity, context, and iteration
3. Zero-shot and few-shot prompting
4. Role-playing prompts
5. Common pitfalls and hallucinations
6. Debugging prompts and iterative refinement
7. Generating simple functions with AI
8. Context injection and RAG
9. Structured output (JSON, XML)
10. Prompt chaining
11. Negative prompting
12. Refinement without test cases
13. Meta-prompting
14. Prompting for code optimisation
15. Chain-of-thought and tree-of-thought
16. Putting it all together

What is a prompt, really?

- A prompt is the *only* input the model receives from you.

What is a prompt, really?

- A prompt is the *only* input the model receives from you.
- Everything that influences the answer must be in the prompt: instructions, context, examples, constraints, output format.

What is a prompt, really?

- A prompt is the *only* input the model receives from you.
- Everything that influences the answer must be in the prompt: instructions, context, examples, constraints, output format.
- The model has no access to your project, your files, or your intent – unless the copilot puts them into the prompt for you.

What is a prompt, really?

- A prompt is the *only* input the model receives from you.
- Everything that influences the answer must be in the prompt: instructions, context, examples, constraints, output format.
- The model has no access to your project, your files, or your intent – unless the copilot puts them into the prompt for you.
- Prompt engineering is the discipline of producing prompts that *reliably* yield the output *you want*.

The four parts of a useful coding prompt

Instruction
("what to do")

The four parts of a useful coding prompt

Instruction
("what to do")

Context
("what to assume")

The four parts of a useful coding prompt

Instruction
("what to do")

Context
("what to assume")

Examples
("what good looks like")

The four parts of a useful coding prompt

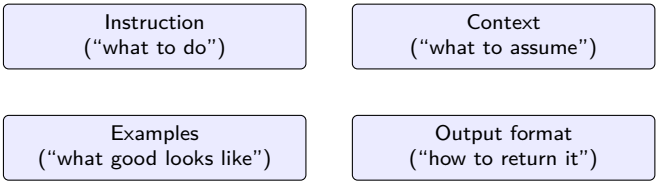
Instruction
(“what to do”)

Context
(“what to assume”)

Examples
(“what good looks like”)

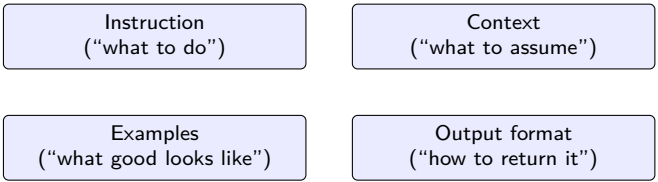
Output format
(“how to return it”)

The four parts of a useful coding prompt



- Missing any of the four invites the model to guess.

The four parts of a useful coding prompt



- Missing any of the four invites the model to guess.
- A model never says “I don’t know”; it fills the gap with something plausible.

A vague prompt vs. a structured one

Vague:

```
Write a function to sort a list.
```

Sort what? Ascending? Java or Python? In-place or returning a new list?

A vague prompt vs. a structured one

Vague:

```
Write a function to sort a list.
```

Sort what? Ascending? Java or Python? In-place or returning a new list?

Structured:

```
Write a Java method:
  signature: public static int[] sortAsc(int[] xs)
  behaviour: return a new array of xs sorted in non-decreasing order;
             xs must not be mutated.
  constraints: do not use any java.util classes.
  output: just the method body, no commentary.
```

The second prompt almost cannot be misinterpreted.

A vague prompt vs. a structured one

Vague:

```
Write a function to sort a list.
```

Sort what? Ascending? Java or Python? In-place or returning a new list?

Structured:

```
Write a Java method:
  signature: public static int[] sortAsc(int[] xs)
  behaviour: return a new array of xs sorted in non-decreasing order;
             xs must not be mutated.
  constraints: do not use any java.util classes.
  output: just the method body, no commentary.
```

The second prompt almost cannot be misinterpreted. Or can it be?

System / user / assistant – the chat protocol

- Most chat APIs split the prompt into three roles:

System / user / assistant – the chat protocol

- Most chat APIs split the prompt into three roles:
  - **System**: persistent instructions (“You are a senior Java reviewer. . . ”).

System / user / assistant – the chat protocol

- Most chat APIs split the prompt into three roles:
  - **System**: persistent instructions ("You are a senior Java reviewer. . . ").
  - **User**: each turn's request.

System / user / assistant – the chat protocol

- Most chat APIs split the prompt into three roles:
  - **System**: persistent instructions ("You are a senior Java reviewer. . . ").
  - **User**: each turn's request.
  - **Assistant**: the model's previous replies (sent back as memory).

System / user / assistant – the chat protocol

- Most chat APIs split the prompt into three roles:
  - **System**: persistent instructions ("You are a senior Java reviewer. . . ").
  - **User**: each turn's request.
  - **Assistant**: the model's previous replies (sent back as memory).
- For inline copilots, the IDE supplies a hidden system prompt; you only edit the user turn implicitly through your code and comments.

System / user / assistant – the chat protocol

- Most chat APIs split the prompt into three roles:
  - **System**: persistent instructions ("You are a senior Java reviewer. . . ").
  - **User**: each turn's request.
  - **Assistant**: the model's previous replies (sent back as memory).
- For inline copilots, the IDE supplies a hidden system prompt; you only edit the user turn implicitly through your code and comments.
- Knowing the role boundaries is essential (Module 6).

Three levers you almost always have

- **Specificity**: turn an open-ended request into a precise one.

Three levers you almost always have

- **Specificity**: turn an open-ended request into a precise one.
- **Context**: paste in the surrounding code, the relevant types, the failing test.

Three levers you almost always have

- **Specificity**: turn an open-ended request into a precise one.
- **Context**: paste in the surrounding code, the relevant types, the failing test.
- **Iteration**: when the answer is wrong, narrow the gap one step at a time.

Three levers you almost always have

- **Specificity**: turn an open-ended request into a precise one.
- **Context**: paste in the surrounding code, the relevant types, the failing test.
- **Iteration**: when the answer is wrong, narrow the gap one step at a time.

Three levers you almost always have

- **Specificity**: turn an open-ended request into a precise one.
- **Context**: paste in the surrounding code, the relevant types, the failing test.
- **Iteration**: when the answer is wrong, narrow the gap one step at a time.

These three habits do more for output quality than any new model release.

Specificity in practice

Less specific:

```
Optimise this method for speed.
```

Specificity in practice

Less specific:

```
Optimise this method for speed.
```

More specific:

```
Reduce the asymptotic complexity of this method from O(n^2) to O(n log n)
without changing its public signature or its observable behaviour
on the provided test cases. Keep it pure Java (no external libraries).
```

Specificity in practice

Less specific:

```
Optimise this method for speed.
```

More specific:

```
Reduce the asymptotic complexity of this method from O(n^2) to O(n log n)
without changing its public signature or its observable behaviour
on the provided test cases. Keep it pure Java (no external libraries).
```

Notice the constraints we added: complexity target, behaviour preservation, no new dependencies.

Context: paste the right surrounding code

Suppose you ask the model to “fix this NullPointerException”. Helpful context to include:

Context: paste the right surrounding code

Suppose you ask the model to "fix this NullPointerException". Helpful context to include:

- The exact stack trace (top 5 frames are usually enough).

Context: paste the right surrounding code

Suppose you ask the model to "fix this NullPointerException". Helpful context to include:

- The exact stack trace (top 5 frames are usually enough).
- The method where the NPE was thrown.

Context: paste the right surrounding code

Suppose you ask the model to "fix this NullPointerException". Helpful context to include:

- The exact stack trace (top 5 frames are usually enough).
- The method where the NPE was thrown.
- Any class fields it touches.

Context: paste the right surrounding code

Suppose you ask the model to "fix this NullPointerException". Helpful context to include:

- The exact stack trace (top 5 frames are usually enough).
- The method where the NPE was thrown.
- Any class fields it touches.
- The test or main method that triggers it.

Context: paste the right surrounding code

Suppose you ask the model to "fix this NullPointerException". Helpful context to include:

- The exact stack trace (top 5 frames are usually enough).
- The method where the NPE was thrown.
- Any class fields it touches.
- The test or main method that triggers it.

Context: paste the right surrounding code

Suppose you ask the model to "fix this NullPointerException". Helpful context to include:

- The exact stack trace (top 5 frames are usually enough).
- The method where the NPE was thrown.
- Any class fields it touches.
- The test or main method that triggers it.

Avoid pasting your entire repository. The model has a context window; bigger is not always better.

Iteration: name the gap

- First response wrong? Don't just say "no, try again".

Iteration: name the gap

- First response wrong? Don't just say "no, try again".
- Say *what* was wrong and *what would be right*:

Iteration: name the gap

- First response wrong? Don't just say "no, try again".
- Say *what* was wrong and *what would be right*:
  - "You used a `HashMap`, but the order of insertion matters here. Use a `LinkedHashMap` instead."

Iteration: name the gap

- First response wrong? Don't just say "no, try again".
- Say *what* was wrong and *what would be right*:
  - "You used a `HashMap`, but the order of insertion matters here. Use a `LinkedHashMap` instead."
  - "Your method modifies the input array; it must return a new array."

Iteration: name the gap

- First response wrong? Don't just say "no, try again".
- Say *what* was wrong and *what would be right*:
  - "You used a `HashMap`, but the order of insertion matters here. Use a `LinkedHashMap` instead."
  - "Your method modifies the input array; it must return a new array."
  - "The complexity is still $O(n^2)$; the inner loop is unnecessary."

Iteration: name the gap

- First response wrong? Don't just say "no, try again".
- Say *what* was wrong and *what would be right*:
  - "You used a `HashMap`, but the order of insertion matters here. Use a `LinkedHashMap` instead."
  - "Your method modifies the input array; it must return a new array."
  - "The complexity is still $O(n^2)$; the inner loop is unnecessary."
- Each iteration should close a measurable gap, not just re-roll the dice.

An iteration in two turns

Turn 1:

```
Write a Java method to remove duplicates from a list of strings.
```

An iteration in two turns

Turn 1:

```
Write a Java method to remove duplicates from a list of strings.
```

Model produces a version using `HashSet` that does not preserve order.

An iteration in two turns

Turn 1:

```
Write a Java method to remove duplicates from a list of strings.
```

Model produces a version using `HashSet` that does not preserve order.

Turn 2:

```
Good, but preserve the original order of first occurrence.
The output must contain each unique string in the order it first appeared.
```

An iteration in two turns

Turn 1:

```
Write a Java method to remove duplicates from a list of strings.
```

Model produces a version using `HashSet` that does not preserve order.

Turn 2:

```
Good, but preserve the original order of first occurrence.
The output must contain each unique string in the order it first appeared.
```

Now the model switches to `LinkedHashSet` or an explicit ordered-iteration approach.

Zero-shot, one-shot, few-shot – definitions

- **Zero-shot**: "Do X." No examples in the prompt.

Examples teach the model the *format* and the *style* of the answer you want, not just the task.

Zero-shot, one-shot, few-shot – definitions

- **Zero-shot**: "Do X." No examples in the prompt.
- **One-shot**: "Do X. Here is one example."

Examples teach the model the *format* and the *style* of the answer you want, not just the task.

Zero-shot, one-shot, few-shot – definitions

- **Zero-shot**: "Do X." No examples in the prompt.
- **One-shot**: "Do X. Here is one example."
- **Few-shot**: "Do X. Here are 2–5 examples."

Examples teach the model the *format* and the *style* of the answer you want, not just the task.

Zero-shot example

```
Write a Java method that returns the longest common prefix of two strings.
Signature: public static String lcp(String a, String b).
```

Works for well-known tasks. Fails when the task is unusual or has implicit conventions.

Zero-shot example

```
Write a Java method that returns the longest common prefix of two strings.
Signature: public static String lcp(String a, String b).
```

Works for well-known tasks. Fails when the task is unusual or has implicit conventions.

Where zero-shot goes wrong:

Zero-shot example

```
Write a Java method that returns the longest common prefix of two strings.
Signature: public static String lcp(String a, String b).
```

Works for well-known tasks. Fails when the task is unusual or has implicit conventions.

Where zero-shot goes wrong:

- "Format an Indian phone number." Zero-shot returns `+91 98XXX XXXXX`; your codebase expects `098XX-XXXXXX`. Wrong convention, no way to guess.

Zero-shot example

```
Write a Java method that returns the longest common prefix of two strings.
Signature: public static String lcp(String a, String b).
```

Works for well-known tasks. Fails when the task is unusual or has implicit conventions.

Where zero-shot goes wrong:

- "Format an Indian phone number." Zero-shot returns `+91 98XXX XXXXX`; your codebase expects `098XX-XXXXXX`. Wrong convention, no way to guess.
- "Convert a status enum to a label using our internal `LabelPolicy`." The model has never seen `LabelPolicy`, so it fabricates a plausible one.

Few-shot example (input/output pairs)

```
Convert each Java enum value name to a human-readable label.
Examples:
  IN_PROGRESS -> "In progress"
  PAID_IN_FULL -> "Paid in full"
  CANCELLED -> "Cancelled"

Now convert: ORDER_SHIPPED, REFUND_REQUESTED, ON_HOLD_BY_FRAUD_CHECK
```

The pattern (snake-shouty $\rightarrow$ sentence case) is taught implicitly by the examples; you did not have to state the rule.

Few-shot example (input/output pairs)

```
Convert each Java enum value name to a human-readable label.
Examples:
  IN_PROGRESS -> "In progress"
  PAID_IN_FULL -> "Paid in full"
  CANCELLED -> "Cancelled"

Now convert: ORDER_SHIPPED, REFUND_REQUESTED, ON_HOLD_BY_FRAUD_CHECK
```

The pattern (snake-shouty $\rightarrow$ sentence case) is taught implicitly by the examples; you did not have to state the rule. Where few-shot goes wrong:

Few-shot example (input/output pairs)

```
Convert each Java enum value name to a human-readable label.
Examples:
  IN_PROGRESS -> "In progress"
  PAID_IN_FULL -> "Paid in full"
  CANCELLED -> "Cancelled"

Now convert: ORDER_SHIPPED, REFUND_REQUESTED, ON_HOLD_BY_FRAUD_CHECK
```

The pattern (snake-shouty $\rightarrow$ sentence case) is taught implicitly by the examples; you did not have to state the rule. Where few-shot goes wrong:

- Contradictory examples: one shows `ON_HOLD` $\rightarrow$ "On hold", another shows the same input mapping to "on-hold". The model picks arbitrarily.

Few-shot example (input/output pairs)

```
Convert each Java enum value name to a human-readable label.
Examples:
  IN_PROGRESS -> "In progress"
  PAID_IN_FULL -> "Paid in full"
  CANCELLED -> "Cancelled"

Now convert: ORDER_SHIPPED, REFUND_REQUESTED, ON_HOLD_BY_FRAUD_CHECK
```

The pattern (snake-shouty $\rightarrow$ sentence case) is taught implicitly by the examples; you did not have to state the rule. Where few-shot goes wrong:

- Contradictory examples: one shows `ON_HOLD` $\rightarrow$ "On hold", another shows the same input mapping to "on-hold". The model picks arbitrarily.
- Examples too narrow: all training pairs are two words, so the model stumbles on `ON_HOLD_BY_FRAUD_CHECK`.

Few-shot example (input/output pairs)

```
Convert each Java enum value name to a human-readable label.
Examples:
  IN_PROGRESS -> "In progress"
  PAID_IN_FULL -> "Paid in full"
  CANCELLED -> "Cancelled"

Now convert: ORDER_SHIPPED, REFUND_REQUESTED, ON_HOLD_BY_FRAUD_CHECK
```

The pattern (snake-shouty $\rightarrow$ sentence case) is taught implicitly by the examples; you did not have to state the rule. Where few-shot goes wrong:

- Contradictory examples: one shows `ON_HOLD` $\rightarrow$ "On hold", another shows the same input mapping to "on-hold". The model picks arbitrarily.
- Examples too narrow: all training pairs are two words, so the model stumbles on `ON_HOLD_BY_FRAUD_CHECK`.
- Examples that leak the answer: including a pair whose input is exactly the target teaches memorisation, not the rule you wanted.

When does few-shot beat zero-shot?

- When the output format is unusual.

When does zero-shot win? When the task is so common in the training data that the model already knows the conventions (sorting, palindrome, fibonacci, basic CRUD).

When does few-shot beat zero-shot?

- When the output format is unusual.
- When there are several plausible interpretations of the task.

When does zero-shot win? When the task is so common in the training data that the model already knows the conventions (sorting, palindrome, fibonacci, basic CRUD).

When does few-shot beat zero-shot?

- When the output format is unusual.
- When there are several plausible interpretations of the task.
- When the task is a transformation you can demonstrate but not easily verbalise.

When does zero-shot win? When the task is so common in the training data that the model already knows the conventions (sorting, palindrome, fibonacci, basic CRUD).

Choosing your few-shot examples

- Pick examples that *span* the input space (easy + tricky + edge).

Choosing your few-shot examples

- Pick examples that *span* the input space (easy + tricky + edge).
- Avoid examples that contradict each other.

Choosing your few-shot examples

- Pick examples that *span* the input space (easy + tricky + edge).
- Avoid examples that contradict each other.
- Avoid examples that leak the answer to the actual question.

Choosing your few-shot examples

- Pick examples that *span* the input space (easy + tricky + edge).
- Avoid examples that contradict each other.
- Avoid examples that leak the answer to the actual question.
- Order matters: the last example is often the most influential.

In-context learning – what's happening under the hood

- The model is not training on your examples.

In-context learning – what's happening under the hood

- The model is not training on your examples.
- It is reading them as part of the context and using its pre-trained capability to extrapolate.

In-context learning – what's happening under the hood

- The model is not training on your examples.
- It is reading them as part of the context and using its pre-trained capability to extrapolate.
- This means: more examples is not always better. After a few, returns diminish; tokens are spent without much gain.

In-context learning – what's happening under the hood

- The model is not training on your examples.
- It is reading them as part of the context and using its pre-trained capability to extrapolate.
- This means: more examples is not always better. After a few, returns diminish; tokens are spent without much gain.
- Rule of thumb: 2–5 examples for most coding tasks.

The role trick

- "You are a security auditor looking for OWASP[1] Top-10 issues."

[1] OWASP = Open Worldwide Application Security Project

The role trick

- "You are a security auditor looking for OWASP[1] Top-10 issues."
- "You are a senior Java engineer reviewing a junior's pull request."

[1] OWASP = Open Worldwide Application Security Project

The role trick

- "You are a security auditor looking for OWASP[1] Top-10 issues."
- "You are a senior Java engineer reviewing a junior's pull request."
- "You are a teaching assistant who never gives the answer directly."

[1] OWASP = Open Worldwide Application Security Project

The role trick

- "You are a security auditor looking for OWASP[1] Top-10 issues."
- "You are a senior Java engineer reviewing a junior's pull request."
- "You are a teaching assistant who never gives the answer directly."
- The same model behaves very differently with different roles, even if the task is identical.

[1] OWASP = Open Worldwide Application Security Project

Without a role

```
Review this method. (paste of a 30-line Java method)
```

Likely answer: a polite, generic “looks fine, maybe consider X” response.

With a role:

```
You are a senior Java engineer doing a serious code review.
Be specific. Quote line numbers. Comment on:
  - correctness, including edge cases,
  - thread safety,
  - error handling,
  - readability and naming.
Review the method below. (paste)
```

Likely answer: a detailed, structured review.

Why roles work

- Training data contains many “senior engineer reviewing” style texts.

Why roles work

- Training data contains many “senior engineer reviewing” style texts.
- Specifying the role pulls the model towards that distribution.

Why roles work

- Training data contains many “senior engineer reviewing” style texts.
- Specifying the role pulls the model towards that distribution.
- This is not magic; it is a statistical bias toward a writing style.

Why roles work

- Training data contains many “senior engineer reviewing” style texts.
- Specifying the role pulls the model towards that distribution.
- This is not magic; it is a statistical bias toward a writing style.
- It also helps you decide *how* to read the answer: a “security auditor” answer should be checked for completeness; a “junior’s draft” answer should be edited.

Pitfall: roles do not grant abilities

- Saying “You are an expert proof assistant” does not make the model produce correct proofs.

Pitfall: roles do not grant abilities

- Saying “You are an expert proof assistant” does not make the model produce correct proofs.
- Saying “You are a cryptographer” does not make the model write secure crypto code.

Pitfall: roles do not grant abilities

- Saying “You are an expert proof assistant” does not make the model produce correct proofs.
- Saying “You are a cryptographer” does not make the model write secure crypto code.
- A role nudges *style and emphasis*, not underlying capability.

Pitfall: roles do not grant abilities

- Saying “You are an expert proof assistant” does not make the model produce correct proofs.
- Saying “You are a cryptographer” does not make the model write secure crypto code.
- A role nudges *style and emphasis*, not underlying capability.
- Treat role prompts as a writing-style switch, not a competence upgrade.

The taxonomy of bad outputs

- **Hallucinations**: fact-like things that are not true (non-existent APIs, fake quotes).

The taxonomy of bad outputs

- **Hallucinations**: fact-like things that are not true (non-existent APIs, fake quotes).
- **Confabulations**: plausible-but-wrong reasoning chains.

The taxonomy of bad outputs

- **Hallucinations**: fact-like things that are not true (non-existent APIs, fake quotes).
- **Confabulations**: plausible-but-wrong reasoning chains.
- **Drift**: the answer slowly stops matching the question over a long chat.

The taxonomy of bad outputs

- **Hallucinations**: fact-like things that are not true (non-existent APIs, fake quotes).
- **Confabulations**: plausible-but-wrong reasoning chains.
- **Drift**: the answer slowly stops matching the question over a long chat.
- **Sycophancy**: agreeing with whatever you just said, even if it is wrong.

The taxonomy of bad outputs

- **Hallucinations**: fact-like things that are not true (non-existent APIs, fake quotes).
- **Confabulations**: plausible-but-wrong reasoning chains.
- **Drift**: the answer slowly stops matching the question over a long chat.
- **Sycophancy**: agreeing with whatever you just said, even if it is wrong.
- **Format breaks**: ignoring the structured-output instruction.

Common Java-side hallucinations

- Non-existent methods.

Common Java-side hallucinations

- Non-existent methods.
- Questions about untrained date.

Common Java-side hallucinations

- Non-existent methods.
- Questions about untrained date.
- Wrong package for a real class.

Common Java-side hallucinations

- Non-existent methods.
- Questions about untrained date.
- Wrong package for a real class.
- Imagined library versions.

Common Java-side hallucinations

- Non-existent methods.
- Questions about untrained date.
- Wrong package for a real class.
- Imagined library versions.
- Confidently wrong advice on language semantics.

Common Java-side hallucinations

- Non-existent methods.
- Questions about untrained date.
- Wrong package for a real class.
- Imagined library versions.
- Confidently wrong advice on language semantics.
- . . .

Common Java-side hallucinations

- Non-existent methods.
- Questions about untrained date.
- Wrong package for a real class.
- Imagined library versions.
- Confidently wrong advice on language semantics.
- . . .

Common Java-side hallucinations

- Non-existent methods.
- Questions about untrained date.
- Wrong package for a real class.
- Imagined library versions.
- Confidently wrong advice on language semantics.
- ...

Use any AI assistant and derive Java code with hallucination

Common Java-side hallucinations

- Non-existent methods.
- Questions about untrained date.
- Wrong package for a real class.
- Imagined library versions.
- Confidently wrong advice on language semantics.
- ...

Use any AI assistant and derive Java code with hallucination

Hallucinations in year XXXX may not remain so in year YYYY

Hallucination example

- Checking validity of credit card number.

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?
- First digit $= 4$?

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?
- First digit $= 4$?
- Luhn's algorithm (by Hans Peter Luhn, IBM)

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?
- First digit $= 4$?
- Luhn's algorithm (by Hans Peter Luhn, IBM)
  - Double alternate digits, starting from the 2nd one on the right.

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?
- First digit = 4?
- Luhn's algorithm (by Hans Peter Luhn, IBM)
  - Double alternate digits, starting from the 2nd one on the right.
    - **8**, 2, **6**, 2, **6**, 2, **8**, 2, **8**, 2, **8**, 2, **8**, 2, **10**, 3

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?
- First digit $= 4$?
- Luhn's algorithm (by Hans Peter Luhn, IBM)
  - Double alternate digits, starting from the 2nd one on the right.
    - **8**, 2, **6**, 2, **6**, 2, **8**, 2, **8**, 2, **8**, 2, **8**, 2, **10**, 3
  - Handle numbers > 9

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?
- First digit $= 4$?
- Luhn's algorithm (by Hans Peter Luhn, IBM)
  - Double alternate digits, starting from the 2nd one on the right.
    - **8**, 2, **6**, 2, **6**, 2, **8**, 2, **8**, 2, **8**, 2, **8**, 2, **10**, 3
  - Handle numbers > 9
    - 8, 2, 6, 2, 6, 2, 8, 2, 8, 2, 8, 2, 8, 2, **1**, 3

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?
- First digit $= 4$?
- Luhn's algorithm (by Hans Peter Luhn, IBM)
  - Double alternate digits, starting from the 2nd one on the right.
    - **8**, 2, **6**, 2, **6**, 2, **8**, 2, **8**, 2, **8**, 2, **8**, 2, **10**, 3
  - Handle numbers > 9
    - 8, 2, 6, 2, 6, 2, 8, 2, 8, 2, 8, 2, 8, 2, **1**, 3
  - Sum the digits.

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?
- First digit $= 4$?
- Luhn's algorithm (by Hans Peter Luhn, IBM)
  - Double alternate digits, starting from the 2nd one on the right.
    - **8**, 2, **6**, 2, **6**, 2, **8**, 2, **8**, 2, **8**, 2, **8**, 2, **10**, 3
  - Handle numbers > 9
    - 8, 2, 6, 2, 6, 2, 8, 2, 8, 2, 8, 2, 8, 2, **1**, 3
  - Sum the digits.
    - 70

Hallucination.java

Hallucination example

- Checking validity of credit card number.
- 4, 2, 3, 2, 3, 2, 4, 2, 4, 2, 4, 2, 4, 2, 5, 0
- 13 or 16 digits?
- First digit $= 4$?
- Luhn's algorithm (by Hans Peter Luhn, IBM)
  - Double alternate digits, starting from the 2nd one on the right.
    - **8**, 2, **6**, 2, **6**, 2, **8**, 2, **8**, 2, **8**, 2, **8**, 2, **10**, 3
  - Handle numbers > 9
    - 8, 2, 6, 2, 6, 2, 8, 2, 8, 2, 8, 2, 8, 2, **1**, 3
  - Sum the digits.
    - 70
  - sum % 10 $==$ 0?

Hallucination.java

Why hallucinations happen

- The model is trained to output the most probable next token, not the most truthful.

Why hallucinations happen

- The model is trained to output the most probable next token, not the most truthful.
- If the right answer never appeared in training, the model still has to output *something*.

Why hallucinations happen

- The model is trained to output the most probable next token, not the most truthful.
- If the right answer never appeared in training, the model still has to output *something*.
- That "something" is the closest plausible-looking neighbour.

Why hallucinations happen

- The model is trained to output the most probable next token, not the most truthful.
- If the right answer never appeared in training, the model still has to output *something*.
- That "something" is the closest plausible-looking neighbour.
- Fine-tuning helps but does not eliminate the problem.

Defences against hallucination

- Compile and run. The compiler catches dead method references and missing imports.

Defences against hallucination

- Compile and run. The compiler catches dead method references and missing imports.
- Give the model the relevant docs / signatures. “Use only methods from this list.”

Defences against hallucination

- Compile and run. The compiler catches dead method references and missing imports.
- Give the model the relevant docs / signatures. “Use only methods from this list.”
- Ask the model to cite or quote rather than paraphrase.

Defences against hallucination

- Compile and run. The compiler catches dead method references and missing imports.
- Give the model the relevant docs / signatures. “Use only methods from this list.”
- Ask the model to cite or quote rather than paraphrase.
- Use models that support tool calls; e.g., let the model run a search before answering.

Defences against hallucination

- Compile and run. The compiler catches dead method references and missing imports.
- Give the model the relevant docs / signatures. “Use only methods from this list.”
- Ask the model to cite or quote rather than paraphrase.
- Use models that support tool calls; e.g., let the model run a search before answering.
- Use negative prompting (next section) to forbid deprecated or non-existent APIs.

Treat the prompt as your real program

- When the model is wrong, it is tempting to “blame the model”.

Treat the prompt as your real program

- When the model is wrong, it is tempting to “blame the model”.
- Often the prompt is the bug: vague instruction, missing context, contradictory examples.

Treat the prompt as your real program

- When the model is wrong, it is tempting to “blame the model”.
- Often the prompt is the bug: vague instruction, missing context, contradictory examples.
- Debugging a prompt is just like debugging code: change one thing at a time, observe the effect, keep going.

The minimal-pair technique

- Save two prompt versions that differ by exactly one element.

The minimal-pair technique

- Save two prompt versions that differ by exactly one element.
- Run both, compare outputs.

The minimal-pair technique

- Save two prompt versions that differ by exactly one element.
- Run both, compare outputs.
- Now you know what the change does.

The minimal-pair technique

- Save two prompt versions that differ by exactly one element.
- Run both, compare outputs.
- Now you know what the change does.
- This is how you build intuition about your model's quirks.

An example of minimal pairing

Prompt A:

```
Write a Java method to compute factorial(n).
```

Model returns a recursive implementation (overflows on large n).

An example of minimal pairing

Prompt A:

```
Write a Java method to compute factorial(n).
```

Model returns a recursive implementation (overflows on large n).

Prompt B:

```
Write a Java method to compute factorial(n) where n can be up to 50.
Use BigInteger.
```

Model returns a `BigInteger` implementation.

An example of minimal pairing

Prompt A:

```
Write a Java method to compute factorial(n).
```

Model returns a recursive implementation (overflows on large n).

Prompt B:

```
Write a Java method to compute factorial(n) where n can be up to 50.
Use BigInteger.
```

Model returns a `BigInteger` implementation.

Prompt C:

```
Write a Java method to compute factorial(n) where n can be up to 50.
Use BigInteger. Don't use recursion.
```

Model returns an iterative `BigInteger` implementation.

Logging your prompt iterations

- Keep a small notebook or markdown file per task: prompt v1, response v1, problem identified, prompt v2 ...

Logging your prompt iterations

- Keep a small notebook or markdown file per task: prompt v1, response v1, problem identified, prompt v2 . . .
- Over a semester you will accumulate a personal cookbook of prompt patterns that work.

Logging your prompt iterations

- Keep a small notebook or markdown file per task: prompt v1, response v1, problem identified, prompt v2 . . .
- Over a semester you will accumulate a personal cookbook of prompt patterns that work.
- This is a real engineering artefact – treat it that way.

The sweet spot

- Small, well-specified, deterministic functions are where AI assistants are at their best.

The sweet spot

- Small, well-specified, deterministic functions are where AI assistants are at their best.
- Examples: sort, search, parse a date, validate an email, convert units.

The sweet spot

- Small, well-specified, deterministic functions are where AI assistants are at their best.
- Examples: sort, search, parse a date, validate an email, convert units.
- These tasks have:

The sweet spot

- Small, well-specified, deterministic functions are where AI assistants are at their best.
- Examples: sort, search, parse a date, validate an email, convert units.
- These tasks have:
  - lots of training-data examples,

The sweet spot

- Small, well-specified, deterministic functions are where AI assistants are at their best.
- Examples: sort, search, parse a date, validate an email, convert units.
- These tasks have:
  - lots of training-data examples,
  - clear correctness criteria,

The sweet spot

- Small, well-specified, deterministic functions are where AI assistants are at their best.
- Examples: sort, search, parse a date, validate an email, convert units.
- These tasks have:
  - lots of training-data examples,
  - clear correctness criteria,
  - small code size that fits in any context window.

Anatomy of a good “small function” prompt

```
Write a Java method:
  signature: public static boolean isValidPan(String s)
  behaviour:
    - returns true iff s is a valid Indian PAN: 5 uppercase letters,
      4 digits, 1 uppercase letter, in that order. Total length 10.
    - returns false if s is null or has any other length.
  constraints: no external libraries; pure Java.
  output: just the method body inside the signature.
```

Notice: explicit format, explicit error case, explicit no-library constraint.

Verifying a small generated function

- Write 5 tests *before* accepting the function: 3 happy + 2 edge.

Verifying a small generated function

- Write 5 tests *before* accepting the function: 3 happy + 2 edge.
- Run all 5.

Verifying a small generated function

- Write 5 tests *before* accepting the function: 3 happy + 2 edge.
- Run all 5.
- Only then merge.

Verifying a small generated function

- Write 5 tests *before* accepting the function: 3 happy + 2 edge.
- Run all 5.
- Only then merge.
- If you cannot describe 5 tests, you do not understand the function well enough to ship it.

Why naive prompts fail on big codebases

- The model knows what is in your prompt.

Why naive prompts fail on big codebases

- The model knows what is in your prompt.
- Your codebase is *not* in your prompt by default.

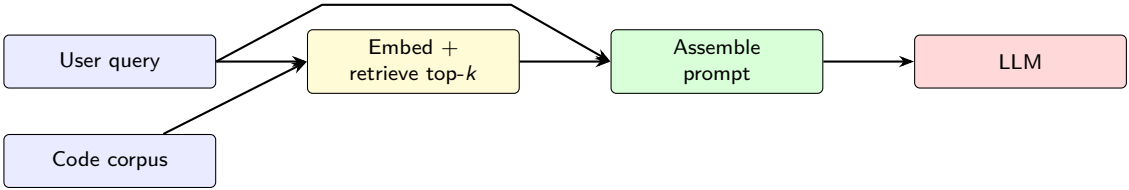
Why naive prompts fail on big codebases

- The model knows what is in your prompt.
- Your codebase is *not* in your prompt by default.
- Pasting the whole repo is impossible – the context window is too small.

Why naive prompts fail on big codebases

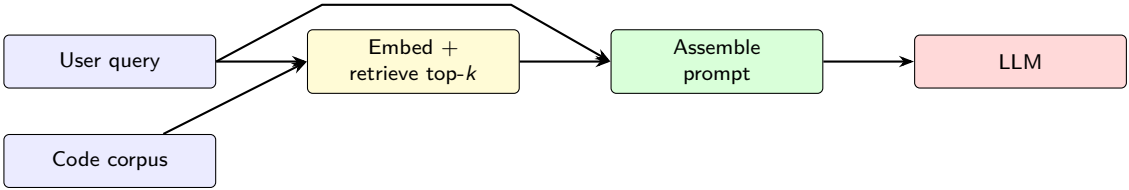
- The model knows what is in your prompt.
- Your codebase is *not* in your prompt by default.
- Pasting the whole repo is impossible – the context window is too small.
- Solution: retrieve only the chunks of code that are relevant to the question, and put those into the prompt.

Retrieval-Augmented Generation (RAG)



- Retrieval finds the relevant chunks; the prompt template stuffs them in alongside the query.

Retrieval-Augmented Generation (RAG)



- Retrieval finds the relevant chunks; the prompt template stuffs them in alongside the query.
- The LLM then "reads" them and answers as if they had been in its prompt all along.

RAG for code repositories

- The “documents” are Java files (or smaller chunks: methods, classes).

RAG for code repositories

- The “documents” are Java files (or smaller chunks: methods, classes).
- Each chunk is embedded into a vector and indexed.

RAG for code repositories

- The “documents” are Java files (or smaller chunks: methods, classes).
- Each chunk is embedded into a vector and indexed.
- Queries are embedded; nearest neighbours are retrieved.

RAG for code repositories

- The “documents” are Java files (or smaller chunks: methods, classes).
- Each chunk is embedded into a vector and indexed.
- Queries are embedded; nearest neighbours are retrieved.
- The chunks are inserted into the prompt with delimiters and a note: “Here is some relevant code. Use it as ground truth.”

RAG pitfalls to know now

- Retrieving the wrong chunks gives a confidently wrong answer.

RAG pitfalls to know now

- Retrieving the wrong chunks gives a confidently wrong answer.
- Embeddings can miss textual or structural matches (semantic similarity $\neq$ behavioural relevance).

RAG pitfalls to know now

- Retrieving the wrong chunks gives a confidently wrong answer.
- Embeddings can miss textual or structural matches (semantic similarity $\neq$ behavioural relevance).
- Chunk boundaries matter: a method split across two chunks may be useless.

RAG pitfalls to know now

- Retrieving the wrong chunks gives a confidently wrong answer.
- Embeddings can miss textual or structural matches (semantic similarity $\neq$ behavioural relevance).
- Chunk boundaries matter: a method split across two chunks may be useless.
- Stale indexes: if the code changed but the index did not, the model is working from a fossil.

RAG pitfalls to know now

- Retrieving the wrong chunks gives a confidently wrong answer.
- Embeddings can miss textual or structural matches (semantic similarity $\neq$ behavioural relevance).
- Chunk boundaries matter: a method split across two chunks may be useless.
- Stale indexes: if the code changed but the index did not, the model is working from a fossil.
- We will return to RAG in detail in Module 6.

Why ask for structured output?

- You want to use the output in code, not show it to a human.

Why ask for structured output?

- You want to use the output in code, not show it to a human.
- Structured output (JSON, XML, CSV) parses cleanly.

Why ask for structured output?

- You want to use the output in code, not show it to a human.
- Structured output (JSON, XML, CSV) parses cleanly.
- Free-form prose answers force you to write regex parsers.

Why ask for structured output?

- You want to use the output in code, not show it to a human.
- Structured output (JSON, XML, CSV) parses cleanly.
- Free-form prose answers force you to write regex parsers.
- Most modern chat APIs offer a “JSON mode” or “response schema” parameter.

Asking for JSON, the right way

```
Extract from the following Java stack trace:
  - the exception type,
  - the message,
  - the topmost user-code frame (file, class, method, line).

Return ONLY a JSON object with these keys:
  "exceptionType", "message", "topUserFrame":
    { "file", "class", "method", "line" }
No prose. No code fences. No leading or trailing text.

Stack trace:
<<paste here>>
```

Defensive JSON parsing (Java side)

```
ObjectMapper m = new ObjectMapper();
JsonNode n;
try {
    n = m.readTree(modelOutput);
} catch (JacksonException e) {
    // Model emitted something that is not JSON. Retry once
    // with: "Your last reply was not valid JSON. Return only JSON."
    n = m.readTree(retryModel(modelOutput, prompt));
}
String exType = n.path("exceptionType").asText();
```

Build the retry into your application: it makes the system robust to occasional format breaks.

Structured output for XML / CSV

- Same principle: tell the model exactly which elements / columns, in which order.

Structured output for XML / CSV

- Same principle: tell the model exactly which elements / columns, in which order.
- XML often needs an example (few-shot) because schemas are richer than JSON's.

Structured output for XML / CSV

- Same principle: tell the model exactly which elements / columns, in which order.
- XML often needs an example (few-shot) because schemas are richer than JSON's.
- CSV: tell the model the exact column order and "one row per record, no header repeated".

When one prompt isn't enough

- Big tasks should not be one big prompt.

When one prompt isn't enough

- Big tasks should not be one big prompt.
- Break them into sub-tasks; let the model handle each one in turn.

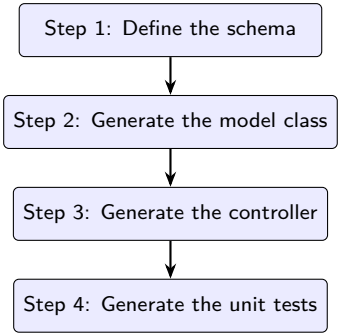
When one prompt isn't enough

- Big tasks should not be one big prompt.
- Break them into sub-tasks; let the model handle each one in turn.
- Each step's output feeds the next step's input.

When one prompt isn't enough

- Big tasks should not be one big prompt.
- Break them into sub-tasks; let the model handle each one in turn.
- Each step's output feeds the next step's input.
- This is just "compose small, well-tested functions" applied to prompts.

Example: building a small REST endpoint



Each step can be inspected and corrected before the next runs.

Why chaining beats monolithic prompts

- Each sub-step has a smaller search space, so the model is more accurate.

Why chaining beats monolithic prompts

- Each sub-step has a smaller search space, so the model is more accurate.
- Failures are easier to localise.

Why chaining beats monolithic prompts

- Each sub-step has a smaller search space, so the model is more accurate.
- Failures are easier to localise.
- You can mix tools: one step might use an LLM, another a deterministic script.

Why chaining beats monolithic prompts

- Each sub-step has a smaller search space, so the model is more accurate.
- Failures are easier to localise.
- You can mix tools: one step might use an LLM, another a deterministic script.
- Total token cost is often *lower* than one giant prompt, because the model does not repeat itself.

Chaining pitfalls

- Error propagation: if step 2 hallucinates a class name, step 3 will use it.

Chaining pitfalls

- Error propagation: if step 2 hallucinates a class name, step 3 will use it.
- State management: each step must know what previous steps decided.

Chaining pitfalls

- Error propagation: if step 2 hallucinates a class name, step 3 will use it.
- State management: each step must know what previous steps decided.
- Latency: k steps means k API calls.

Chaining pitfalls

- Error propagation: if step 2 hallucinates a class name, step 3 will use it.
- State management: each step must know what previous steps decided.
- Latency: k steps means k API calls.
- Mitigation: add cheap validators between steps (compile, lint, schema-check).

Telling the model what *not* to do

- Models tend to reach for what they have seen most often.

Telling the model what *not* to do

- Models tend to reach for what they have seen most often.
- Often that includes deprecated APIs (`java.util.Date`, `StringTokenizer`, raw types).

Telling the model what *not* to do

- Models tend to reach for what they have seen most often.
- Often that includes deprecated APIs (`java.util.Date`, `StringTokenizer`, raw types).
- A negative prompt forbids these explicitly.

A negative prompt

```
Write a Java method to parse "2026-07-15" into a date object.

Do NOT use:
  - java.util.Date
  - java.text.SimpleDateFormat
  - the joda-time library
Use only java.time (LocalDate / DateTimeFormatter).
```

Without the negative list, many models will reach for `SimpleDateFormat` on autopilot.

Where negative prompts especially help

- Java: `java.util.Date` vs `java.time`.

Where negative prompts especially help

- Java: `java.util.Date` vs `java.time`.
- Logging: `System.out.println` vs SLF4J.

Where negative prompts especially help

- Java: `java.util.Date` vs `java.time`.
- Logging: `System.out.println` vs SLF4J.
- Concurrency: `Thread.stop()` (deprecated) vs cooperative interruption.

Where negative prompts especially help

- Java: `java.util.Date` vs `java.time`.
- Logging: `System.out.println` vs SLF4J.
- Concurrency: `Thread.stop()` (deprecated) vs cooperative interruption.
- Web: legacy synchronous APIs vs modern reactive ones.

Where negative prompts especially help

- Java: `java.util.Date` vs `java.time`.
- Logging: `System.out.println` vs SLF4J.
- Concurrency: `Thread.stop()` (deprecated) vs cooperative interruption.
- Web: legacy synchronous APIs vs modern reactive ones.
- Anywhere your project has chosen a modern stack and you want the model to stay there.

What if there are no tests yet?

- Sometimes you are exploring; you don't have a test harness.

What if there are no tests yet?

- Sometimes you are exploring; you don't have a test harness.
- You still need a way to refine the model's output without "vibes".

What if there are no tests yet?

- Sometimes you are exploring; you don't have a test harness.
- You still need a way to refine the model's output without "vibes".
- Strategies:

What if there are no tests yet?

- Sometimes you are exploring; you don't have a test harness.
- You still need a way to refine the model's output without "vibes".
- Strategies:
  - ask the model to enumerate edge cases and address each,

What if there are no tests yet?

- Sometimes you are exploring; you don't have a test harness.
- You still need a way to refine the model's output without "vibes".
- Strategies:
  - ask the model to enumerate edge cases and address each,
  - ask for two different implementations and compare,

What if there are no tests yet?

- Sometimes you are exploring; you don't have a test harness.
- You still need a way to refine the model's output without "vibes".
- Strategies:
  - ask the model to enumerate edge cases and address each,
  - ask for two different implementations and compare,
  - ask the model to act as both author and reviewer.

The “critic” prompt

```
Step A: Write the method.
Step B: Now imagine you are a strict reviewer. List five concrete
        ways this method could be wrong or fragile.
Step C: Rewrite the method addressing each of those issues.
```

This compresses two iterations into one prompt and pulls some self-review out of the model.

Caveats on self-critique

- Self-critique tends to be shallow: the model finds plausible-sounding issues, not necessarily the real ones.

Caveats on self-critique

- Self-critique tends to be shallow: the model finds plausible-sounding issues, not necessarily the real ones.
- Calibration is still bad: it will not flag a deep bug confidently.

Caveats on self-critique

- Self-critique tends to be shallow: the model finds plausible-sounding issues, not necessarily the real ones.
- Calibration is still bad: it will not flag a deep bug confidently.
- Use as a complement to tests, not a substitute.

Prompts that write prompts

- Meta-prompting: one LLM produces a prompt that you (or another LLM) will use.

Prompts that write prompts

- Meta-prompting: one LLM produces a prompt that you (or another LLM) will use.
- Useful when you don't know what a good prompt for a task looks like.

Prompts that write prompts

- Meta-prompting: one LLM produces a prompt that you (or another LLM) will use.
- Useful when you don't know what a good prompt for a task looks like.
- Example: "Write the best possible prompt for asking a Java code assistant to refactor a class for testability."

A meta-prompt

```
You are a prompt engineer. The task is:
  "Take a piece of legacy Java code and produce a list of refactorings
   that will improve testability without changing public behaviour."
Produce a single prompt I can send to a coding assistant
to accomplish this task. Make it specific, include the desired output
format, and add a negative-prompt clause listing common bad refactorings
to avoid.
```

Now you have a polished prompt template you can reuse.

Pipelines of meta-prompts

- "Generate the prompt, then run it, then critique the output, then rewrite the prompt."

Pipelines of meta-prompts

- "Generate the prompt, then run it, then critique the output, then rewrite the prompt."
- This is where "agentic" workflows begin.

Pipelines of meta-prompts

- "Generate the prompt, then run it, then critique the output, then rewrite the prompt."
- This is where "agentic" workflows begin.
- Be careful: every additional LLM call is a chance for an error to enter and propagate.

Optimisation prompts done well

- Always specify the dimension being optimised: time complexity, memory, readability, lines of code.

Optimisation prompts done well

- Always specify the dimension being optimised: time complexity, memory, readability, lines of code.
- Always specify the constraint: “without changing public behaviour”, “preserve thread-safety”.

Optimisation prompts done well

- Always specify the dimension being optimised: time complexity, memory, readability, lines of code.
- Always specify the constraint: “without changing public behaviour”, “preserve thread-safety”.
- Always include a baseline measurement if you have one.

A complete optimisation prompt

```
The method below has measured runtime ~120 ms on the benchmark
suite at the bottom. The hot path is the nested loop on lines 10-18.

Rewrite the method to reduce runtime to under 30 ms on the same
benchmark, without changing:
  - the public signature,
  - the observable output on any input,
  - the use of pure Java (no native libraries).

After the code, briefly explain the change in 2-3 sentences.
```

Pitfalls in optimisation prompts

- The model may “optimise” by deleting input validation. Forbid this explicitly.

Pitfalls in optimisation prompts

- The model may “optimise” by deleting input validation. Forbid this explicitly.
- It may suggest unsafe parallelism. State whether multi-threading is allowed.

Pitfalls in optimisation prompts

- The model may “optimise” by deleting input validation. Forbid this explicitly.
- It may suggest unsafe parallelism. State whether multi-threading is allowed.
- It may quietly change the API. Re-state the signature in the constraint list.

Pitfalls in optimisation prompts

- The model may “optimise” by deleting input validation. Forbid this explicitly.
- It may suggest unsafe parallelism. State whether multi-threading is allowed.
- It may quietly change the API. Re-state the signature in the constraint list.
- Benchmarks lie if you don’t warm up the JVM; explain your measurement methodology.

Chain-of-thought (CoT)

- Ask the model to think *step by step* before giving the final answer.

Chain-of-thought (CoT)

- Ask the model to think *step by step* before giving the final answer.
- For non-trivial problems this often improves accuracy.

Chain-of-thought (CoT)

- Ask the model to think *step by step* before giving the final answer.
- For non-trivial problems this often improves accuracy.
- Reason: the model uses its own output as scratch space; longer outputs mean more "thinking" tokens.

CoT in practice

```
Solve this Java problem step by step.

Problem: implement a method to validate a Sudoku board.

Output format:
  1) Restate the problem in your own words.
  2) Describe the checking strategy in three or four steps.
  3) Now write the Java method that follows that strategy.
  4) Briefly explain why each row/column/box check is correct.
```

Tree-of-thought (ToT)

- Instead of one chain, the model is asked to explore *multiple* reasoning paths in parallel.

Tree-of-thought (ToT)

- Instead of one chain, the model is asked to explore *multiple* reasoning paths in parallel.
- Each path scores how promising it is; the system commits to the best one.

Tree-of-thought (ToT)

- Instead of one chain, the model is asked to explore *multiple* reasoning paths in parallel.
- Each path scores how promising it is; the system commits to the best one.
- Useful for branching problems: planning, search, multi-step refactors.

Tree-of-thought (ToT)

- Instead of one chain, the model is asked to explore *multiple* reasoning paths in parallel.
- Each path scores how promising it is; the system commits to the best one.
- Useful for branching problems: planning, search, multi-step refactors.
- In practice often implemented externally as "run the model several times, pick the best output".

When CoT/ToT are worth the tokens

- Problems with multi-step reasoning (parsing, planning, algorithm design).

When to skip them: boilerplate, syntactic conversions, look-ups.

When CoT/ToT are worth the tokens

- Problems with multi-step reasoning (parsing, planning, algorithm design).
- Problems where the final answer alone is hard to verify.

When to skip them: boilerplate, syntactic conversions, look-ups.

When CoT/ToT are worth the tokens

- Problems with multi-step reasoning (parsing, planning, algorithm design).
- Problems where the final answer alone is hard to verify.
- Problems where a small mistake early on cascades.

When to skip them: boilerplate, syntactic conversions, look-ups.

A warning on hidden reasoning

- Newer models hide their chain-of-thought from you.

A warning on hidden reasoning

- Newer models hide their chain-of-thought from you.
- This is sometimes for product reasons, sometimes for safety.

A warning on hidden reasoning

- Newer models hide their chain-of-thought from you.
- This is sometimes for product reasons, sometimes for safety.
- Be aware: the visible output is not the full “thought”.

A warning on hidden reasoning

- Newer models hide their chain-of-thought from you.
- This is sometimes for product reasons, sometimes for safety.
- Be aware: the visible output is not the full “thought”.
- Where you can see the chain, read it – it often shows you why the final answer is wrong.

A checklist for any non-trivial coding prompt

1. State the task precisely (specificity).

A checklist for any non-trivial coding prompt

1. State the task precisely (specificity).
2. Provide relevant context (signatures, types, failing test).

A checklist for any non-trivial coding prompt

1. State the task precisely (specificity).
2. Provide relevant context (signatures, types, failing test).
3. Provide 0–5 examples (few-shot only when format/style is unusual).

A checklist for any non-trivial coding prompt

1. State the task precisely (specificity).
2. Provide relevant context (signatures, types, failing test).
3. Provide 0–5 examples (few-shot only when format/style is unusual).
4. Pick a role only if it sharpens the answer style.

A checklist for any non-trivial coding prompt

1. State the task precisely (specificity).
2. Provide relevant context (signatures, types, failing test).
3. Provide 0–5 examples (few-shot only when format/style is unusual).
4. Pick a role only if it sharpens the answer style.
5. Forbid known bad patterns (negative prompts).

A checklist for any non-trivial coding prompt

1. State the task precisely (specificity).
2. Provide relevant context (signatures, types, failing test).
3. Provide 0–5 examples (few-shot only when format/style is unusual).
4. Pick a role only if it sharpens the answer style.
5. Forbid known bad patterns (negative prompts).
6. Specify the output format exactly (JSON/code/prose; with or without comments).

A checklist for any non-trivial coding prompt

1. State the task precisely (specificity).
2. Provide relevant context (signatures, types, failing test).
3. Provide 0–5 examples (few-shot only when format/style is unusual).
4. Pick a role only if it sharpens the answer style.
5. Forbid known bad patterns (negative prompts).
6. Specify the output format exactly (JSON/code/prose; with or without comments).
7. Ask for reasoning steps if the task is multi-step (CoT).

A checklist for any non-trivial coding prompt

1. State the task precisely (specificity).
2. Provide relevant context (signatures, types, failing test).
3. Provide 0–5 examples (few-shot only when format/style is unusual).
4. Pick a role only if it sharpens the answer style.
5. Forbid known bad patterns (negative prompts).
6. Specify the output format exactly (JSON/code/prose; with or without comments).
7. Ask for reasoning steps if the task is multi-step (CoT).
8. After the answer: verify (compile, run, test, read).

Common antipatterns recap

- “Make it better.”

Each of these says nothing the model can act on.

Common antipatterns recap

- “Make it better.”
- “Optimise this.”

Each of these says nothing the model can act on.

Common antipatterns recap

- "Make it better."
- "Optimise this."
- "Why doesn't this work?" (without the error message).

Each of these says nothing the model can act on.

Common antipatterns recap

- "Make it better."
- "Optimise this."
- "Why doesn't this work?" (without the error message).
- "Use the latest library."

Each of these says nothing the model can act on.

Common antipatterns recap

- "Make it better."
- "Optimise this."
- "Why doesn't this work?" (without the error message).
- "Use the latest library."
- "You're wrong. Try again."

Each of these says nothing the model can act on.

Common antipatterns recap

- "Make it better."
- "Optimise this."
- "Why doesn't this work?" (without the error message).
- "Use the latest library."
- "You're wrong. Try again."
- "Are you sure?"

Each of these says nothing the model can act on.

References used in this module

- Schulhoff et al., *The Prompt Report: A Systematic Survey of Prompt Engineering Techniques*, arXiv:2406.06608, 2024.
- Wei et al., *Chain-of-Thought Prompting Elicits Reasoning in Large Language Models*, NeurIPS 2022.
- Yao et al., *Tree of Thoughts: Deliberate Problem Solving with Large Language Models*, NeurIPS 2023.
- Lewis et al., *Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks*, NeurIPS 2020.
- El Amri, *LLM Prompt Engineering For Developers*, 2023.

Questions?