

**הנדסת פרומפטים: מתמטיקה, שיטה וביצוע  
למערכות IA מתקדמות**

**Prompt Engineering: Mathematics, Method, and Execution  
for Advanced AI Systems**

ד"ר יורם סגל

כל הזכויות שמורות - © Dr. Segal Yoram

December 2025

גרסה 1.0

## תוכן העניינים

6	<b>1 יסודות תיאורטיים של הנדסת פרומפטים</b>
6	1.1 הגדרות יסוד: Fundamental Definitions
7	1.2 מבוא: המהפכה השקטה של השפה
7	1.3 הפרדוקס של העדרויות: The Paradox of Absences
7	1.4 משפט Wiio למודלי שפה גדולים
8	1.5 מרחב ההסתברות המותנית: Conditional Probability Space
8	1.6 יסודות מתמטיים: Mathematical Foundations
9	1.6.1 איך מחשבים את $H$ – אי-ודאות התשובה בהינתן הפרומפט
9	1.6.2 דוגמה מספרית: פרומפט טוב מול פרומפט גרוע
10	1.6.3 המסקנה המעשית
11	1.6.4 מורכבות (Perplexity): כמה המודל מופתע?
11	1.6.5 מבחן הניחוש: אינטואיציה פשוטה
12	1.6.6 איך מחשבים מורכבות? שלושה צעדים
12	1.6.7 דוגמה מספרית: "החתול ישב"
12	1.6.8 השוואה: משפט הגיוני מול משפט מוזר
13	1.6.9 הקשר למשוואה 1
14	1.7 השוואת מטרות פרומפטים: Prompt Objectives Comparison
16	1.8 ויזואליזציה של זרימת מידע: Information Flow Visualization
16	1.9 דוגמאות עשה ואל תעשה: Do and Don't Examples
18	1.10 סיכום הפרק: Chapter Summary
19	<b>2 מתודולוגיה מערכתית לתכנון פרומפטים</b>
19	2.1 עקרון הפרומפט האטומי: Atomic Prompting Principle
20	2.1.1 פירוש הטבלה: ארבעה שלבים בתהליך העבודה
21	2.2 השוואת פרומפטים: Prompt Diffing
22	2.3 כלל שלושת הפרומפטים: The 3-Prompt Rule
23	2.4 נוסחאות מתמטיות: Mathematical Formulas
24	2.5 דוגמאות עשה ואל תעשה: Do and Don't Examples
26	2.6 סיכום הפרק: Chapter Summary
27	<b>3 תבניות פרומפט מתקדמות וארכיטקטורות</b>
27	3.1 מבוא: תפקיד המתכנת בעידן ההסקה
27	3.1.1 הפיגום החיצוני
27	3.1.2 שני מישורים של בקרה
28	3.1.3 המסקנה החשובה
28	3.2 Chain-of-Thought: שרשרת מחשבה
28	3.2.1 הגדרה פורמלית

28	הסבר נרטיבי	3.2.2
29	דוגמה פשוטה	3.2.3
29	גרסה מתקדמת: Chain-of-Thought++	3.2.4
29	מהות השיטה המתקדמת	3.2.5
30	הצבעת עקביות עצמית	3.2.6
30	ReAct: שילוב חשיבה ופעולה	3.3
30	הגדרה פורמלית	3.3.1
30	הסבר נרטיבי	3.3.2
31	דוגמה פשוטה	3.3.3
31	גרסה מתקדמת: ReAct 2.0	3.3.4
31	מעבר לפרדיגמה חדשה	3.3.5
31	ארכיטקטורת הזרימה	3.3.6
33	Tree-of-Thoughts: עץ מחשבות	3.4
33	הגדרה פורמלית	3.4.1
33	הסבר נרטיבי	3.4.2
33	דוגמה פשוטה	3.4.3
33	גרסה מתקדמת: Tree-of-Thoughts עם תקציב כפוי	3.4.4
34	חקירה מבנית	3.4.5
34	גיוס דינמי	3.4.6
34	הנחיית תפקידים: Role-Based Prompting	3.5
36	למידה מדוגמאות: אופטימיזציה של Few-Shot	3.6
36	השוואה מערכתית בין תבניות	3.7
36	דוגמאות עשה ואל תעשה: Do and Don't Examples	3.8
38	חלון ההקשר בפעולה: Context Window Mechanics	3.9
38	Chain-of-Thought: צמיחת החלון	3.9.1
39	ReAct: עצירות וחזרות	3.9.2
39	Tree-of-Thoughts: ענפים מקבילים	3.9.3
40	מה קורה כשהחלון נגמר?	3.9.4
40	סיכום הפרק: Chapter Summary	3.10
41	<b>4 הנדסת פרומפטים לסוכני IA</b>	
41	מבוא: מהמענה לפעולה	4.1
41	ממשק סוכן-מחשב: Agent-Computer Interface (ACI)	4.2
42	תבנית ReAct	4.2.1
42	ארכיטקטורות סוכנים: Agent Architectures	4.3
43	מעקות בטיחות כפרומפטים: Guardrails as Prompts	4.4
43	מטא-פרומפטינג: Meta-Prompting	4.5
44	נוסחאות מתמטיות: Mathematical Formulas	4.6
45	שימוש בכלים: Tool-Use	4.7

45	Do and Don't Examples: דוגמאות עשה ואל תעשה:	4.8
46	Chapter Summary: סיכום הפרק:	4.9
47	<b>5 הערכה, אופטימיזציה והטמעה בייצור</b>	
47	מבוא: מהמעבדה לייצור	5.1
47	Prompt Versioning: ניהול גרסאות פרומפטים:	5.2
48	Statistical Significance: מובהקות סטטיסטית:	5.3
48	A/B Testing Methodology: מתודולוגיית בדיקות A/B:	5.4
49	Drift Detection: זיהוי סחיפה:	5.5
50	CI/CD Pipeline: צינור CI/CD לפרומפטים:	5.6
51	Cost-Accuracy Optimization: אופטימיזציית עלות-דיוק:	5.7
52	Do and Don't Examples: דוגמאות עשה ואל תעשה:	5.8
53	Chapter Summary: סיכום הפרק:	5.9
54	מקורות בעברית	5.10
	English References	5.11
		54

## רשימת האיורים

16	זרימת מידע דרך שכבות Attention בטרנספורמר	1
17	משטח דיוק כפונקציה של אורך פרומפט וטמפרטורה (מערך נתונים GSM8K)	2
	השוואה בין פרומפטים אטומיים לפרומפט מונוליתי. הגישה האטומית	3
20	משיגה דיוק כולל של 89% לעומת 72% בגישה המונוליתית.	
	שיפור איטרטיבי: דיוק עולה והזיות יורדות לאורך ארבע איטרציות של	4
23	שיפור פרומפט.	
24	מחזור החיים של הנדסת פרומפטים: מהגדרת משימה ועד פריסה בייצור.	5
32	ארכיטקטורת ReAct 2.0: מעגל Act-Observe-Reflect-Adapt	6
35	עץ מחשבות עם גיזום: ענפים עם ציון נמוך נגזמים	7
50	תוצאות ניסוי A/B: השוואת גרסאות פרומפט לאורך 7 ימים	8
51	צינור CI/CD לפרומפטים: משלב הפיתוח ועד לייצור	9

## רשימת הטבלאות

15	השוואת מטרות פרומפטים לפי מדדי מידע	1
15	מדדי אופטימליות פרומפט לדגמים שונים (2025)	2
20	פירוק פרומפט למשימת ניתוח קוד	3
22	מדדי שיפור איטרציה-אחר-איטרציה	4
36	אופטימיזציית few-shot learning	5
37	השוואת תבניות פרומפטים לפי סיבוכיות	6
38	חלון הקשר ב-CoT – שלב 1	7

38	חלון הקשר ב-CoT - שלב 2	8
39	חלון הקשר ב-ReAct - מחזור שלם	9
39	חלון הקשר ב-ToT - הערכת ענפים	10
42	ארכיטקטורות פרומפטים לסוכנים	11
43	סוגי מעקות בטיחות ופרומפטים	12
49	מתודולוגיית בדיקות A/B לפרומפטים	13
52	מדדי הערכה לפרומפטים בייצור	14

# 1 יסודות תיאורטיים של הנדסת פרומפטים

## מטרות הלמידה

בסיום פרק זה, הקורא יוכל:

- להסביר את מנגנון ה-Attention ואת הקשר שלו ליעילות פרומפטים
- לבנות מודל מידע לתכנון אופטימלי של tokens
- להגדיר פורמלית את מרחב הפרומפטים ופונקציית התשובה
- לנתח את הפרדוקס של העדרויות בתכנון פרומפטים

## 1.1 הגדרות יסוד: Fundamental Definitions

לפני שנצלול לעומק, נגדיר את המושגים המרכזיים.

### מהו פרומפט?

המילה prompt באנגלית פירושה "הנחיה" או "זירוז לפעולה". בעברית, המונח המקביל הוא **הנחיה**. לאורך הספר נשתמש בשני המונחים לסירוגין: **הנחיה** ו-**פרומפט** (בכתיב עברי).

### הגדרה: פרומפט (Prompt)

**הגדרה פורמלית:** פרומפט הוא רצף של טקסט בשפה טבעית המשמש כקלט למודל שפה גדול (LLM), במטרה להנחות את המודל לייצר פלט רצוי.

במילים פשוטות: פרומפט הוא השאלה שואל את המכונה. אבל זה יותר משאלה. זו הדרך שבה אתה מסביר למכונה מה אתה רוצה. פרומפט טוב הוא כמו הוראות ברורות לעובד חדש. פרומפט גרוע הוא כמו לבקש "תעשה משהו טוב".

### מהי הנדסת פרומפטים?

### הגדרה: הנדסת פרומפטים (Prompt Engineering)

**הגדרה פורמלית:** הנדסת פרומפטים היא התהליך השיטתי של עיצוב, מבנה ואופטימיזציה של הנחיות טקסטואליות, במטרה לשפר את איכות הפלט ממודלי בינה מלאכותית גנרטיבית [1].

הגדרה זו מבוססת על סקירה אקדמית מקיפה מ-2024 שניתחה למעלה מ-1500 מאמרים בתחום.

במילים פשוטות: הנדסת פרומפטים היא האמנות והמדע של לדבר עם מכונות. זה לא רק מה שאתה אומר. זה איך אתה אומר. באיזה סדר. עם כמה פרטים. מהנדס פרומפטים טוב מבין את המכונה. הוא יודע מה היא צריכה לשמוע כדי לעבוד נכון.

## למה זה חשוב?

חישבו על זה כך: יש לכם כלי עוצמתי. המפתח לשימוש בו הוא התקשורת. מודלי שפה גדולים יכולים לכתוב קוד, לנתח נתונים, ליצור תוכן. אבל הם עושים רק מה שמבקשים מהם. והבקשה היא הפרומפט.

לכן הנדסת פרומפטים היא לא רק מיומנות טכנית. היא מיומנות תקשורת. היא דורשת חשיבה ברורה, ניסוח מדויק, והבנה של הצד השני. בספר זה נלמד את שניהם: את התיאוריה המתמטית ואת האמנות המעשית.

## 1.2 מבוא: המהפכה השקטה של השפה

לפני 70000 שנה קרה משהו מוזר. בני אדם התחילו לדבר. השפה שינתה הכל. היא אפשרה לנו לשתף רעיונות. היא יצרה תרבויות שלמות. היום אנחנו עומדים בפני מהפכה דומה. מודלי שפה גדולים (Large Language Models) הם הישג טכנולוגי עצום. הם מבינים שפה אנושית. הם יוצרים טקסט חדש. אבל יש בעיה: איך אנחנו מתקשרים איתם? התשובה היא **הנדסת פרומפטים** [1]. פרומפט הוא הוראה בשפה טבעית. הוא אומר למודל מה לעשות. פרומפט טוב מניב תוצאות טובות. פרומפט גרוע מניב שטויות. הפרק הזה מציג את היסודות התיאורטיים של התחום. נלמד מתמטיקה. נבין מדוע פרומפטים עובדים. נגלה את הכללים שמנחים תקשורת יעילה עם מכונות.

## 1.3 הפרדוקס של העדרויות: The Paradox of Absences

### הפרדוקס של העדרויות

פרומפט ממוקד מדי מפספס פתרונות יצירתיים; פרומפט רחב מדי מגביר hallucinations. האתגר הוא למצוא את נקודת האיזון האופטימלית.

כשאנחנו כותבים פרומפט, אנחנו עומדים בפני דילמה. אם הפרומפט ספציפי מדי, המודל מוגבל. הוא לא יכול לחשוב מחוץ לקופסה. אם הפרומפט כללי מדי, המודל מבולבל. הוא עלול להמציא עובדות [2].

מחקרים מראים שהזיות (Hallucinations) קורות ב-5 עד 15 אחוז מהתשובות. הסיבה: המודל ממלא חסר. כשאין מספיק מידע, הוא מנחש [3]. הפתרון הוא איזון. פרומפט טוב נותן מספיק הקשר. לא יותר מדי. לא פחות מדי. זה הפרדוקס של העדרויות: מה שלא נכתב חשוב כמו מה שנכתב.

## 1.4 משפט Wiio למודלי שפה גדולים

## משפט Wiio ל-LLMs

"Communication usually fails, except by accident." כל פרומפט דורש validation דרך גבולות תיאורטיים-מידעיים.

פרופסור אוסמו ויאו הציג חוק פשוט: תקשורת בדרך כלל נכשלת. זה נכון גם למודלי שפה. רוב הפרומפטים לא עובדים כמו שצפינו. הסיבה נעוצה בתורת המידע [4]. כל הודעה מכילה מידע. המידע נמדד ב-bits. כשהמידע בפרומפט לא מספיק, המודל ממלא חסר. כשהמידע מרובה מדי, המודל מתבלבל. האנטרופיה (Entropy) מודדת אי-ודאות. פרומפט טוב מצמצם אי-ודאות. הוא מכווין את המודל לתשובה הנכונה. פרומפט גרוע מגביר אי-ודאות. הוא משאיר את המודל מבולבל.

## 1.5 מרחב ההסתברות המותנית: Conditional Probability Space

### מרחב ההסתברות המותנית

$P(\text{answer}|\text{prompt})$  אינו מונוטוני ב-context length. קיימת נקודת אופטימום שקשורה למורכבות (perplexity) של המודל.

מודלי שפה הם מכונות הסתברותיות. הם מחשבים: מה הסיכוי לתשובה  $y$  בהינתן פרומפט  $x$ ? זו ההסתברות המותנית:  $P(y|x)$ . אבל יש הפתעה. פרומפט ארוך יותר לא תמיד טוב יותר. מחקר חשוב מ-2024 גילה תופעה מעניינת [5]. המודלים שוכחים מידע באמצע הפרומפט. הם זוכרים את ההתחלה והסוף. זה נקרא "Lost in the Middle". אורך הפרומפט משפיע על הדיוק. אבל לא בצורה ליניארית. יש נקודה אופטימלית. פרומפט קצר מדי חסר מידע. פרומפט ארוך מדי מבלבל. איור 2 מציג את הקשר בין אורך, טמפרטורה ודיוק. אפשר לראות את ה-ridge האופטימלי: 300 עד 500 טוקנים בטמפרטורה 0.7.

## 1.6 יסודות מתמטיים: Mathematical Foundations

עכשיו נפרמל את הרעיונות. נגדיר משוואות שמתארות פרומפטים אופטימליים.

### משוואה 1.1: פונקציית אובדן לפרומפטים

$$(1) \quad \mathcal{L}_{prompt} = \alpha \cdot H(y|x) + \beta \cdot \frac{|x|}{C_{max}} + \gamma \cdot \text{Perplexity}(x)$$

המשוואה מגדירה פונקציית אובדן [6]. ככל שהערך נמוך יותר, הפרומפט טוב יותר.

-  $x$  = הפרומפט שכתבנו



-  $y$  = התשובה שאנחנו רוצים

-  $H(y|x)$  = אי-ודאות התשובה בהינתן הפרומפט

-  $C_{max}$  = גודל חלון ההקשר של המודל

-  $\alpha, \beta, \gamma$  = משקלות שקובעים את האיזון

הרכיב הראשון מודד איכות. הרכיב השני מודד יעילות. הרכיב השלישי מודד בהירות.

### 1.6.1 איך מחשבים את $H$ – אי-ודאות התשובה בהינתן הפרומפט

עצרו רגע. ראינו את המשתנה  $H(y|x)$  במשוואה. אבל מה זה אומר בפועל? איך מחשבים "אי-ודאות"? התשובה מגיעה מתורת המידע של שאנון [4]. כשאנחנו שואלים מודל שפה שאלה, הוא לא באמת "יודע" את התשובה. הוא מחשב הסתברויות. לכל מילה אפשרית הוא נותן ציון בין 0 ל-1. המילה עם הציון הגבוה ביותר נבחרת כתשובה. אבל מה קורה כשהמודל לא בטוח? כשכמה מילים מקבלות ציונים דומים? זה בדיוק מה שהאנטרופיה מודדת: כמה המודל "מבולבל".

#### נוסחת האנטרופיה המותנית

$$(2) \quad H(Y|x) = - \sum_{i=1}^n P(y_i|x) \cdot \log_2(P(y_i|x))$$

**מקרא משתנים:**

-  $n$  = מספר המילים האפשריות באוצר המילים של המודל

-  $y_i$  = מילה ספציפית (למשל: "עברית", "אנגלית", "צרפתית")

-  $P(y_i|x)$  = ההסתברות שהמודל נותן למילה  $y_i$  בהינתן הפרומפט  $x$

-  $\log_2$  = לוגריתם בבסיס 2 (התוצאה נמדדת בביטים)

נשמע מסובך? בואו נראה דוגמה פשוטה.

### 1.6.2 דוגמה מספרית: פרומפט טוב מול פרומפט גרוע

נניח שאנחנו מזינים למודל את הפרומפט: "In Israel, people speak the...". המודל צריך לחזות את המילה הבאה.

**תרחיש א': פרומפט ממוקד (מודל בטוח)**

הפרומפט ברור. המודל "יודע" מה התשובה. הוא נותן את ההסתברויות הבאות:

- "Hebrew": 0.9

- "Arabic": 0.1

- כל שאר המילים: 0

נחשב את האנטרופיה:

$$\begin{aligned} H &= -[0.9 \cdot \log_2(0.9) + 0.1 \cdot \log_2(0.1)] \\ &= -[0.9 \cdot (-0.152) + 0.1 \cdot (-3.321)] \\ &= -[-0.137 + (-0.332)] \\ &= 0.47 \text{ bits} \end{aligned}$$

### תוצאה טובה

ערך אנטרופיה נמוך (0.47 ביטים) = המודל בטוח בתשובה שלו. זה מה שאנחנו רוצים. הפרומפט עובד.

### תרחיש ב': פרומפט עמום (מודל מבולבל)

עכשיו נשנה את הפרומפט ל: "The language is...". אין הקשר. על איזו מדינה מדובר? המודל לא יודע.

ההסתברויות מתפזרות:

- "English": 0.5

- "Hebrew": 0.5

נחשב:

$$\begin{aligned} H &= -[0.5 \cdot \log_2(0.5) + 0.5 \cdot \log_2(0.5)] \\ &= -[0.5 \cdot (-1) + 0.5 \cdot (-1)] \\ &= -[-0.5 + (-0.5)] \\ &= 1.0 \text{ bit} \end{aligned}$$

### ▲ שימו לב להבדל

ערך אנטרופיה גבוה (1.0 ביט) = המודל מבולבל. הפרומפט לא עובד. צריך לשפר אותו.

### 1.6.3 המסקנה המעשית

חזרו למשוואה 1. המטרה היא למזער את  $\mathcal{L}_{prompt}$ . הרכיב  $H(y|x)$  הוא חלק מהמשוואה. ככל שהוא קטן יותר, הפרומפט טוב יותר. איך מקטינים את  $H$ ? בדיוק כמו בדוגמה:

- פרומפט ברור וממוקד  $\rightarrow$  המודל בטוח  $\rightarrow H$  נמוך

- פרומפט עמום וכללי  $\rightarrow$  המודל מבולבל  $\rightarrow H$  גבוה

במילים פשוטות:  $H(y|x)$  הוא **מדד המבוכה** של המודל. כשהמודל "יודע" מה לענות, המבוכה נמוך. כשהוא מנחש, המבוכה גבוה. תפקיד מהנדס הפרומפטים הוא להפחית את המבוכה למינימום.

#### 1.6.4 מורכבות (Perplexity): כמה המודל מופתע?

עכשיו נכיר מדד נוסף: **מורכבות** (באנגלית: Perplexity). המילה המקורית פירושה "מבוכה" או "תימהון". וזה בדיוק מה שהמדד מודד. חשבו על זה כך: האנטרופיה  $H$  מודדת מבוכה בביטים. המורכבות מודדת את אותו מבוכה במונחים אחרים: **מספר האפשרויות הסבירות**.

#### 1.6.5 מבחן הניחוש: אינטואיציה פשוטה

דמיינו שאתם מנסים לנחש את המילה הבאה במשפט.

- אם המורכבות היא 1: אתם בטוחים לחלוטין. יש רק אפשרות אחת הגיונית.
  - אם המורכבות היא 6: אתם מבולבלים כאילו הייתם צריכים להטיל קובייה.
  - אם המורכבות היא 100: אתם אבודים לחלוטין. כמו לבחור קלף מחפיסה כפולה.
- ככל שהמספר נמוך יותר, המודל פחות מופתע. ככל שהמספר גבוה יותר, המודל בהלם.

#### נוסחת המורכבות (Perplexity)

$$(3) \quad PP(x) = \sqrt[N]{\frac{1}{P(w_1) \cdot P(w_2) \cdot \dots \cdot P(w_N)}}$$

**מקרא משתנים:**

-  $PP(x)$  = המורכבות של הטקסט  $x$

-  $N$  = מספר המילים בטקסט

-  $w_1, w_2, \dots, w_N$  = המילים בטקסט לפי הסדר

-  $P(w_i)$  = ההסתברות של המילה  $w_i$  בהינתן המילים שלפניה

-  $\sqrt[N]{\phantom{x}}$  = שורש מסדר  $N$  (ממוצע הנדסי)

נשמע מסובך? בואו נפרק את זה לשלבים פשוטים.

### 1.6.6 איך מחשבים מורכבות? שלושה צעדים

**צעד 1: מוצאים הסתברות לכל מילה.** מה הסיכוי שהמילה הזו תופיע, בהינתן המילים שבאו לפנייה?

**צעד 2: מכפילים את כל ההסתברויות.** מקבלים את הסיכוי לכל המשפט כולו.

**צעד 3: מנרמלים.** לוקחים את ההופכי ומוציאים שורש מסדר  $N$ .

### 1.6.7 דוגמה מספרית: "החתול ישב"

נחשב את המורכבות של המשפט הקצר: **"החתול ישב"** (2 מילים).

**צעד 1: מוצאים הסתברויות**

המודל נותן את התחזיות הבאות:

- המילה "החתול" (בתחילת משפט):  $P = 0.01$

- המילה "ישב" (אחרי "החתול"):  $P = 0.1$

למה 0.01? כי יש הרבה מילים שאפשר להתחיל איתן משפט. למה 0.1? כי חתול יכול גם ללכת, לרוץ, לאכול. לשבת זו אפשרות אחת מתוך רבות.

**צעד 2: מכפילים**

$$P(\text{המשפט}) = 0.01 \times 0.1 = 0.001$$

**צעד 3: מחשבים מורכבות**

$$\begin{aligned} PP &= \sqrt[2]{\frac{1}{0.001}} \\ &= \sqrt{1000} \\ &= 31.6 \end{aligned}$$

### מה המספר אומר?

המורכבות היא 31.6. המשמעות: המודל היה מבולבל כאילו היה עליו לבחור מתוך כ-32 מילים אפשריות בכל שלב.

### 1.6.8 השוואה: משפט הגיוני מול משפט מוזר

בואו נראה למה מורכבות נמוכה היא דבר טוב.

**משפט א' (הגיוני): "ירד גשם"**

- הסתברות ל"ירד": 0.05

- הסתברות ל"גשם" (אחרי "ירד"): 0.8 – מאוד צפוי!

$$P(\text{המשפט}) = 0.05 \times 0.8 = 0.04$$

$$PP = \sqrt{\frac{1}{0.04}} = \sqrt{25} = 5$$

**משפט ב' (מוזר): "ירד פיל"**

- הסתברות ל"ירד": 0.05

- הסתברות ל"פיל" (אחרי "ירד"): 0.0001 – מאוד מפתיע!

$$P(\text{המשפט}) = 0.05 \times 0.0001 = 0.000005$$

$$PP = \sqrt{\frac{1}{0.000005}} = \sqrt{200000} = 447$$

### ▲ ההבדל דרמטי

"ירד גשם": מורכבות 5. המודל לא מופתע.

"ירד פיל": מורכבות 447. המודל בהלם.

### 1.6.9 הקשר למשוואה 1

חזרו למשוואה 1. שימו לב לרכיב האחרון:

$$\dots + \gamma \cdot \text{Perplexity}(x) / \text{מורכבות}(x)$$

הרכיב הזה מודד את המורכבות של הפרומפט עצמו. לא של התשובה. של הפרומפט  $x$  שאתם כותבים.

מה קורה אם תכתבו פרומפט עם שגיאות כתיב? עם מילים שלא קשורות? עם ניסוחים מסורבלים? המורכבות תהיה גבוהה. המודל יתקשה "להבין" מה אתם רוצים.

### הכלל הפשוט

פרומפט טוב = מורכבות נמוכה = שפה ברורה, תקנית וצפויה.  
פרומפט גרוע = מורכבות גבוהה = שפה מבולבלת, משובשת או מוזרה.

המשוואה אומרת: כתבו פרומפטים פשוטים וברורים. אל תנסו להרשים את המודל עם שפה מליצית. הוא לא מתרשם. הוא רק מתבלבל.

### משוואה 2.1: מבחן צוואר הבקבוק המידעי

$$(4) \quad I(x; y) \leq \min\{H(x), H(y)\} - \epsilon \cdot \text{len}(x)$$

המשוואה מבוססת על תורת המידע [4]. היא קובעת גבול עליון למידע שיכול לעבור מהפרומפט לתשובה.

-  $I(x; y)$  = המידע ההדדי בין הפרומפט לתשובה

-  $H(x)$  = האנטרופיה של הפרומפט

-  $H(y)$  = האנטרופיה של התשובה

-  $\epsilon$  = קבוע דעיכה עבור פרומפטים ארוכים

אם  $I(x; y)$  נמוך מדי, הפרומפט חסר מידע. אם גבוה מדי, הפרומפט רועש.

### משוואה 3.1: השפעת המיקום

$$(5) \quad P_{correct} = \sigma(\theta_0 + \theta_1 \cdot \text{position}_{instruction} + \theta_2 \cdot \text{position}_{example})$$

המשוואה מבוססת על מחקר Lost in the Middle [5]. היא מראה שהמיקום של המידע בפרומפט משפיע על הדיוק.

-  $\sigma$  = פונקציית סיגמואיד (ממירה לסיכוי בין 0 ל-1)

-  $\theta_0$  = הטיה בסיסית

-  $\theta_1$  = משקל למיקום ההוראה

-  $\theta_2$  = משקל למיקום הדוגמאות

מחקרים מראים:  $\theta_1 > \theta_2$  עבור משימות חשיבה. ההוראה צריכה להיות בהתחלה. עבור משימות יצירה, המגמה הפוכה: הדוגמאות חשובות יותר.

## 1.7 השוואת מטרות פרומפטים: Prompt Objectives Comparison

טבלה 1 מציגה סוגי משימות שונים. לכל משימה יש מאפיינים אחרים. המספרים מבוססים על מחקרים מ-2024 [1].

טבלה 2 משווה בין מודלים שונים. המספרים עדכניים ל-2025 ומבוססים על מחקרי שוק [6].

טבלה 1: השוואת מטרות פרומפטים לפי מדדי מידע

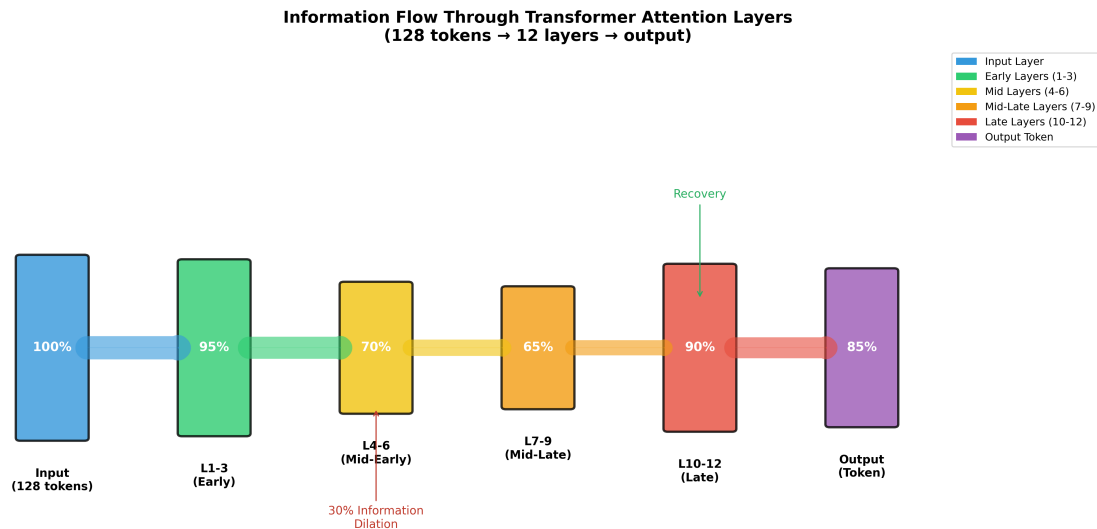
מטרת פרומפט	Entropy	Token Budget	סף דיוק	דוגמה לשימוש
סיווג	< 2 bits	50–100	0.95	סיווג סנטימנט
יצירה	5–7 bits	200–500	0.85	כתיבת תקציר
חשיבה	< 3 bits	400–800	0.90	פתרון מתמטי
קוד	< 4 bits	300–600	0.92	כתיבת פונקציות

טבלה 2: מדדי אופטימליות פרומפט לדגמים שונים (2025)

טמפרטורה מומלצת	עלות	Attention Eff.	אורך אופטימלי	Context	Model
0.6–0.8	\$3/M	0.87	800–1200	200K	Claude 3.5
0.5–0.7	\$5/M	0.91	600–900	128K	GPT-4o
0.4–0.6	\$2.5/M	0.94	1500–3000	1M	Gemini 2.5
0.6–0.9	\$0.8/M	0.85	500–800	128K	Llama 3.1

## 1.8 ויזואליזציה של זרימת מידע: Information Flow Visualization

איך המידע זורם דרך המודל? הטרנספורמר [7] עובד בשכבות. כל שכבה מעבדת את המידע. חלק מהמידע נשמר. חלק אובד. איור 1 מציג את הזרימה. 128 טוקנים נכנסים. הם עוברים 12 שכבות. בסוף יוצא טוקן אחד. שימו לב: בשכבה 6 יש ירידה של 30 אחוז בצפיפות המידע. בשכבה 10 יש התאוששות חלקית.



איור 1: זרימת מידע דרך שכבות Attention בטרנספורמר

איור 2 מציג משטח תלת-ממדי. צירי ה- $X$  ו- $Y$  הם אורך הפרומפט והטמפרטורה. ציר ה- $Z$  הוא הדיקו. אפשר לראות בבירור: יש אזור אופטימלי. מחוץ לאזור הזה, הביצועים יורדים.

## 1.9 דוגמאות עשה ואל תעשה: Do and Don't Examples

הנה דוגמאות מעשיות. למדו מהן איך לכתוב פרומפטים טובים.

✓ עשה:

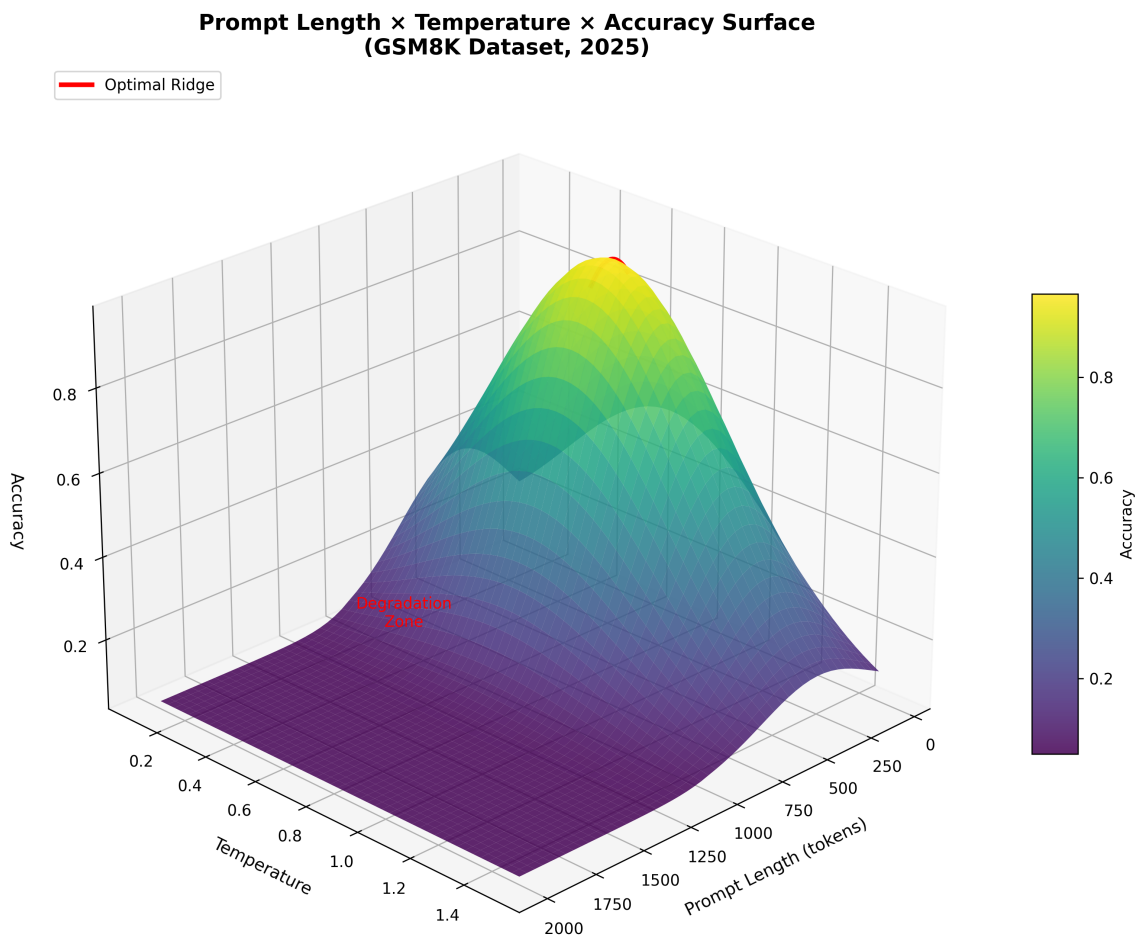
"כתוב פונקציית פייתון שמחשבת את הממוצע המשוקלל של רשימה. השתמש ב-type hints, וכתוב docstring עם examples."

▲ אל תעשה:

"תן לי קוד לחישוב ממוצע."

הפרומפט הטוב ספציפי. הוא מגדיר את השפה. הוא דורש תיעוד. הוא נותן הקשר. הפרומפט הגרוע מעורפל. המודל צריך לנחש מה אנחנו רוצים.





איור 2: משטח דיוק כפונקציה של אורך פרומפט וטמפרטורה (מערך נתונים GSM8K)

✓ עשה:

"הסבר את אלגוריתם Dijkstra. תחילה, הגדר את המשתנים: graph, source, distances. אחרי כל שלב, הצג את ה-state של מערך ה-distances."

▲ אל תעשה:

"מה זה Dijkstra?"

הפרומפט הטוב משתמש בשרשרת חשיבה [8]. הוא מבקש הסבר צעד אחר צעד. הוא מגדיר את המבנה מראש.

✓ עשה:

"תרגם את הטקסט לאנגלית. שמור על רמת שפה פורמלית וטון מקצועי. הטקסט: 'הדוח הכספי מצביע על גידול של 15 אחוז ברווחים.'"

▲ אל תעשה:

"תרגם: 'הדוח הכספי מצביע על גידול של 15 אחוז ברווחים.'"

הפרומפט הטוב מגדיר סגנון. הוא מציין את הרמה הרצויה. הוא נותן הקשר.

✓ עשה:

"השתמש בשרשרת חשיבה: קודם, חשב את השורש הראשון. אחר כך, הוסף אותו לממוצע. הסבר כל צעד."

▲ אל תעשה:

"פתור:  $\text{mean}() + \sqrt{16}$ "

## 1.10 סיכום הפרק: Chapter Summary

בפרק זה למדנו את היסודות התיאורטיים של הנדסת פרומפטים. גילינו את הפרדוקס של העדרויות. פרומפט צריך להיות מאוזן. לא ספציפי מדי. לא כללי מדי. למדנו את משפט וייאו. תקשורת נכשלת בדרך כלל. לכן צריך לבדוק כל פרומפט. הבנו את מרחב ההסתברות. אורך הפרומפט משפיע על הדיוק. יש נקודה אופטימלית. הצגנו שלוש משוואות מתמטיות:

- משוואה 1: פונקציית האובדן לפרומפטים

- משוואה 4: צוואר הבקבוק המידעי

- משוואה 5: השפעת המיקום

בפרק הבא נעבור מתיאוריה לפרקטיקה. נלמד מתודולוגיה מערכתית לתכנון פרומפטים. נראה איך לפרק פרומפטים למרכיבים אטומיים. נבנה מערכת הערכה שיטתית.

## 2 מתודולוגיה מערכתית לתכנון פרומפטים

במאה העשרים, המדענים גילו סוד פשוט. כדי להבין מערכת מורכבת, צריך לפרק אותה לחלקים קטנים. האטום הפך למפתח להבנת החומר. הגן הפך למפתח להבנת החיים. כיום, אנו מגלים שאותו עיקרון פועל גם בתקשורת עם מערכות AI. פרומפט טוב אינו תוצאה של השראה פתאומית. הוא תוצאה של שיטה. פרק זה מציג את השיטה הזו.

### מטרות הלמידה

בסיום פרק זה, הקורא יוכל:

- לפתח framework מדיד לכתיבת פרומפטים
- ליישם atomic prompting ו-prompt decomposition
- לבנות prompt evaluation suite
- להשתמש בכלל שלושת הפרומפטים לאופטימיזציה

### 2.1 עקרון הפרומפט האטומי: Atomic Prompting Principle

#### עקרון הפרומפט האטומי

כל פרומפט חייב לבדוק היפותזה אחת בלבד. פרומפטים מורכבים הם הרכבה של אטומים דרך אופרטורי composition.

מהו אטום? בפיזיקה, אטום הוא החלקיק הקטן ביותר שעדיין שומר על תכונות היסוד. בהנדסת פרומפטים, פרומפט אטומי הוא ההוראה הקטנה ביותר שעדיין מבצעת משימה אחת מוגדרת [9].

בואו נחשוב על משימה מורכבת: "ניתוח קוד". משימה זו כוללת למעשה ארבע משימות נפרדות:

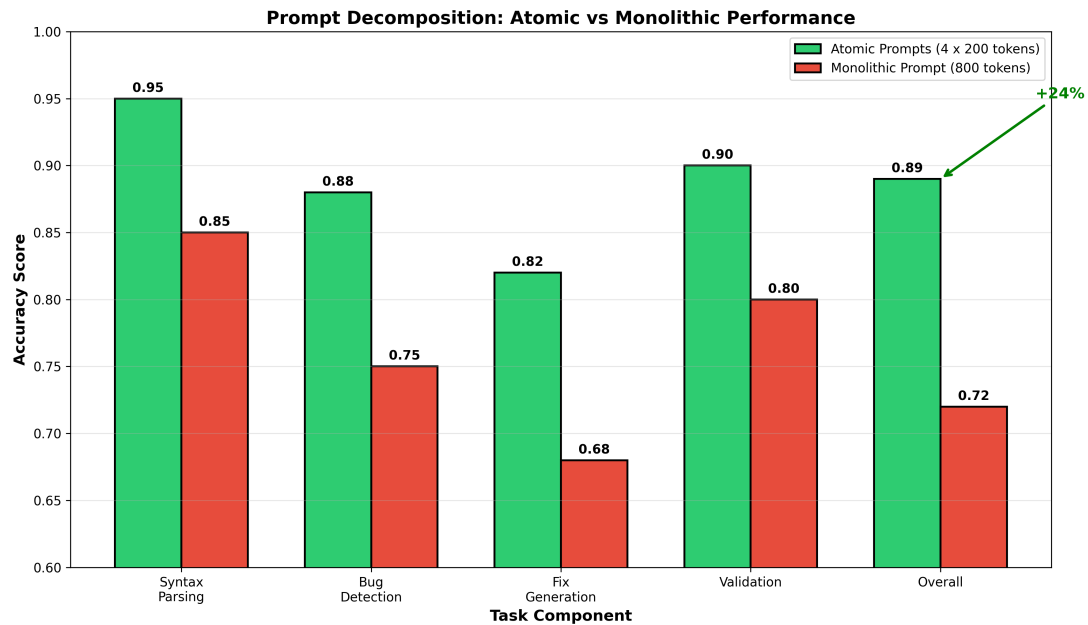
1. ניתוח תחביר (Syntax Parsing)

2. זיהוי באגים (Bug Detection)

3. יצירת תיקון (Fix Generation)

4. אימות התיקון (Validation)

פרומפט מונוליתי מנסה לעשות הכל בבת אחת. פרומפט אטומי מפרק את המשימה לארבעה חלקים. מחקרים מראים שהגישה האטומית משיגה דיוק גבוה יותר [1]. איור 3 מציג את ההבדל.



איור 3: השוואה בין פרומפטים אטומיים לפרומפט מונוליטי. הגישה האטומית משיגה דיוק כולל של 89% לעומת 72% בגישה המונוליטית.

למה זה עובד? התשובה טמונה באופן שבו מודלי שפה מעבדים מידע. כאשר פרומפט מכיל משימות רבות, המודל מחלק את תשומת הלב ביניהן. כאשר פרומפט מכיל משימה אחת, כל תשומת הלב מתמקדת בה [5].

טבלה 3: פירוק פרומפט למשימת ניתוח קוד

כלי נדרש	דיוק צפוי	תקציב	סוג פרומפט	רכיב
AST parser	0.95	100	Zero-shot	ניתוח תחביר
Static analyzer	0.88	200	Few-shot	זיהוי באגים
Code executor	0.82	300	Chain-of-Thought	יצירת תיקון
Test runner	0.90	150	Self-consistency	אימות

### 2.1.1 פירוש הטבלה: ארבעה שלבים בתהליך העבודה

טבלה 3 מציגה תהליך עבודה שלם לפיתוח תוכנה בעזרת AI. כל שורה מייצגת שלב אחר. כל שלב דורש סוג פרומפט שונה וכלי שונה. זוהי אמנות הפירוק לאטומים בפעולה.

**שלב ראשון: ניתוח תחביר** המשימה הראשונה היא הפשוטה ביותר: לבדוק אם הקוד כתוב בצורה תקינה. האם חסר סוגר? האם יש נקודה-פסיק במקום הנכון? שאלות אלה הן

בינאריות. התשובה היא "כן" או "לא".

**סוג הפרומפט:** Zero-shot – פנייה למודל ללא דוגמאות מקדימות. מדוע? בדיקת תחביר היא משימה בסיסית. המודל אינו זקוק לדוגמאות כדי לדעת אם חסר סוגר. הפרומפט יהיה פשוט: "האם הקוד הבא תקין תחבירית?"

**הכלי הנדרש:** AST parser – מנתח עץ תחביר מופשט. כלי זה אינו מבוסס AI. הוא לוקח את הקוד ומנסה לבנות ממנו "עץ" היררכי. אם הבנייה נכשלת, יש שגיאת תחביר. זהו כלי דטרמיניסטי. הוא מדויק ב-100%.

**שלב שני: זיהוי באגים** המשימה השנייה מורכבת יותר. אנחנו מחפשים שגיאות לוגיות. הקוד עובר את בדיקת התחביר, אבל האם הוא עושה מה שהוא צריך לעשות?

**סוג הפרומפט:** Few-shot – פרומפט שכולל 2-3 דוגמאות של קלט ופלט. למשל: "הנה קוד עם באג → הנה הבאג שנמצא". הדוגמאות עוזרות למודל להבין איזה סוג של באגים אנחנו מחפשים. האם אבטחה? ביצועים? לוגיקה?

**הכלי הנדרש:** Static analyzer – מנתח סטטי. כלי שסורק את הקוד מבלי להריץ אותו. הוא דומה לבודק איות, אבל לקוד. כלים מוכרים הם ESLint, Pylint, ו-SonarQube. הם מזהים תבניות בעייתיות ידועות מראש.

**שלב שלישי: יצירת תיקון** המשימה השלישית היא הקשה ביותר. המודל צריך לכתוב קוד מתוקן. זו כבר לא בדיקה. זו יצירה.

**סוג הפרומפט:** Chain-of-Thought – שרשרת חשיבה. אנחנו מנחים את המודל "לחשוב בקול רם". לפרק את הבעיה לשלבים לפני שהוא כותב את הקוד. אם המודל ינסה ישר לפלוט קוד, הוא עלול לטעות. אבל אם הוא קודם יכתוב: "1. הבאג נובע מלולאה אינסופית. 2. צריך להוסיף תנאי עצירה. 3. הנה הקוד המתוקן" – הדיוק יעלה משמעותית [8].

**הכלי הנדרש:** Code executor – מריץ קוד. סביבת Sandbox שבה ניתן להריץ את הקוד שהמודל כתב. זהו המבחן המעשי הראשון. האם הקוד בכלל עובד? האם הוא קורס?

**שלב רביעי: אימות** המשימה האחרונה היא קריטית. התיקון נכתב. הקוד רץ. אבל האם הוא באמת פתר את הבעיה? האם הוא לא יצר בעיות חדשות?

**סוג הפרומפט:** Self-consistency – עקביות עצמית. טכניקה שבה מבקשים מהמודל לייצר את הפתרון מספר פעמים. למשל, 5 פעמים. אז בוחרים את התשובה הנפוצה ביותר. אם המודל הציע את אותו פתרון 4 מתוך 5 פעמים, הסבירות שהוא נכון גבוהה מאוד [10].

**הכלי הנדרש:** Test runner – מריץ בדיקות. כלי שמריץ סט של בדיקות אוטומטיות (Unit Tests) שהוגדרו מראש. הוא נותן את החותמת הסופית: "עבר" או "נכשל". אם הקוד עבר את כל הטסטים, התיקון מאושר.

זהו תהליך מלא. ארבעה שלבים. ארבעה סוגי פרומפט. ארבעה כלים. כל אחד מותאם למשימה הספציפית שלו. זו המהות של גישת הפרומפט האטומי.

## 2.2 השוואת פרומפטים: Prompt Diffing

## Prompt Diffing

השוואת שני פרומפטים על אותו input כדי לזהות מרכיבים קריטיים. כל שינוי מעל 5% ב-accuracy מצריך Root Cause Analysis.

לפני שרופאים מחליטים איזו תרופה יעילה יותר, הם עורכים ניסוי מבוקר. קבוצה אחת מקבלת את התרופה. קבוצה אחרת מקבלת פלצבו. ההבדל בתוצאות מגלה את האפקט האמיתי.

הנדסת פרומפטים דורשת אותה קפדנות מדעית. כאשר משנים פרומפט, צריך למדוד את ההשפעה. זוהי טכניקת Prompt Diffing [11].  
התהליך פשוט:

1. קח פרומפט קיים (baseline)

2. שנה אלמנט אחד בלבד

3. הרץ את שני הפרומפטים על אותם נתונים

4. מדוד את ההבדל בדיוק

אם ההבדל גדול מ-5%, יש לבצע ניתוח שורש הבעיה. מה בשינוי גרם לשיפור או להרעה? התשובה מלמדת אותנו מה באמת חשוב בפרומפט.

טבלה 4 מציגה דוגמה לשיפור איטרטיבי. כל גרסה משפרת את הקודמת.

טבלה 4: מדדי שיפור איטרציה-אחר-איטרציה

איטרציה	גרסה	דיוק	הזיות	השהייה	שינוי
1	v0.1	0.72	0.15	1200ms	-
2	v0.2	0.79	0.12	1350ms	+9.7%
3	v0.3	0.85	0.08	1800ms	+7.6%
4	v0.4	0.89	0.05	2100ms	+4.7%

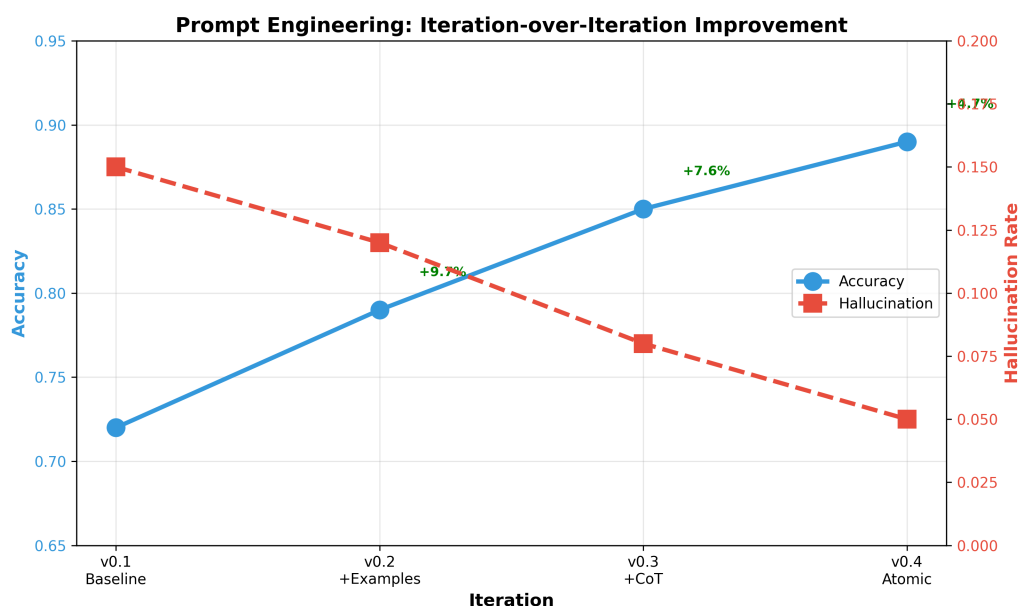
כפי שאיור 4 מראה, הדיוק עולה מ-72% ל-89%. במקביל, שיעור ההזיות יורד מ-15% ל-5%. זהו שיפור משמעותי שנבנה צעד אחר צעד.

## 2.3 כלל שלושת הפרומפטים: The 3-Prompt Rule

### כלל שלושת הפרומפטים

לכל משימה, כתוב שלושה פרומפטים:

- Minimal - 50 tokens



איור 4: שיפור איטרטיבי: דיוק עולה והזיות יורדות לאורך ארבע איטרציות של שיפור פרומפט.

tokens 200 – Balanced –  
tokens 500 – Comprehensive –  
בדוק איזה מודל מגיב הכי טוב לכל אחד.

הסיפור של זהבה ושלושת הדובים מלמד אותנו עיקרון חשוב. לא קטן מדי, לא גדול מדי – בדיוק מתאים. עיקרון זה תקף גם לפרומפטים. מחקרים מ-2025 מראים תופעה מפתיעה: יותר מידע לא תמיד אומר תוצאות טובות יותר [12]. לפעמים פרומפט קצר עובד טוב יותר מפרומפט ארוך. הכל תלוי במשימה ובמודל. לכן, הכלל הוא פשוט: כתוב שלוש גרסאות. בדוק את כולן. בחר את הטובה ביותר. איור 5 מציג את מחזור החיים המלא של הנדסת פרומפטים. שלב ההערכה הוא קריטי: רק פרומפט שעובר את סף הדיוק מגיע לייצור.

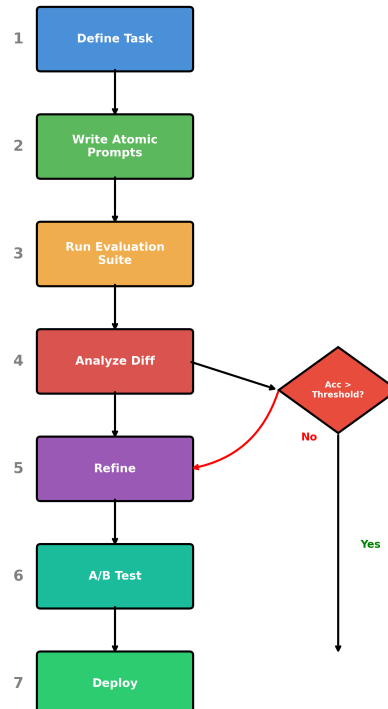
## 2.4 נוסחאות מתמטיות: Mathematical Formulas

כדי למדוד את יעילות הפירוק לפרומפטים אטומיים, אנו משתמשים בציון פירוק פרומפט (Prompt Decomposition Score).

### משוואה 1.2: ציון פירוק פרומפט (PDS)

$$(6) \quad PDS = \frac{\sum_{i=1}^n \text{Acc}(p_i) \cdot \log(\text{len}(p_i))}{\log(\text{len}(p_{full}))}$$

### Prompt Engineering Lifecycle



איור 5: מחזור החיים של הנדסת פרומפטים: מהגדרת משימה ועד פריסה בייצור.

כאשר  $p_i$  = פרומפט אטומי,  $p_{full}$  = פרומפט מונוליטי. ציון  $PDS > 1.15$  מצביע על יתרון .decomposition.

הנוסחה משקללת את הדיוק של כל פרומפט אטומי לפי אורכו. פרומפט קצר עם דיוק גבוה מקבל ציון גבוה יותר. אם הציון הכולל עולה על 1.15, הפירוק לאטומים משתלם. כדי לדעת אם שיפור בדיוק הוא אמיתי ולא מקרי, אנו משתמשים בנוסחת רווח הסמך.

### משוואה 2.2: רווח בר-סמך לשיפור פרומפט

$$(7) \quad \Delta_{\text{significant}} = z \cdot \sqrt{\frac{p_1(1-p_1)}{n_1} + \frac{p_2(1-p_2)}{n_2}}$$

עבור  $z = 1.96$  (רמת סמך 95%), שיפור ב-accuracy נחשב מובהק רק אם  $\Delta > \Delta_{\text{significant}}$ .

לדוגמה: אם בדקנו 1000 דוגמאות, שיפור של 2% בדיוק הוא מובהק סטטיסטית. שיפור של 0.5% עשוי להיות מקרי.

### 2.5 דוגמאות עשה ואל תעשה: Do and Don't Examples

ההבדל בין פרומפט טוב לפרומפט גרוע הוא לעיתים קטן אך משמעותי. הנה דוגמאות מעשיות.



✓ עשה: Atomic Decomposition

"משימה 1: חלץ את כל שמות המשתנים מהקוד הבא. רשום אותם כ-bullet points. קוד: ..."

▲ אל תעשה: Monolithic

"תנתח את הקוד, מצא באגים, תקן אותם, וכתוב דוח."

הפרומפט הראשון מבקש משימה אחת ברורה. הפרומפט השני מבקש ארבע משימות בבת אחת.

✓ עשה: Explicit Steps

"שלב 1: קרא את השאלה. שלב 2: חשב את ה-derivative. שלב 3: הכנס את  $x = 2$ . שלב 4: בדוק את התשובה."

▲ אל תעשה: Implicit

"פתור את השאלה."

הפרומפט הראשון מפרט את הצעדים. הפרומפט השני מניח שהמודל ידע מה לעשות. לפעמים הוא יודע. לפעמים לא.

✓ עשה: Validation Loop

"כתוב פונקציה. אחרי הכתיבה, הרץ אותה עם `input=5`. אם התוצאה  $\neq 15$ , תקן את הלוגיקה."

▲ אל תעשה: No Validation

"כתוב פונקציה."

הפרומפט הראשון כולל מנגנון בדיקה עצמית. הפרומפט השני לא. מחקרים מראים ש-self-validation מפחית טעויות ב-30% [13].

✓ עשה: Diff Analysis

"השווה בין שני הפרומפטים הבאים. הצג הבדלים ב-accuracy, latency, ו-hallucination rate."

▲ אל תעשה: No Comparison

"בדוק איזה פרומפט טוב יותר."

הפרומפט הראשון מגדיר קריטריונים ברורים. הפרומפט השני מותיר את ההחלטה למודל.

## 2.6 סיכום הפרק: Chapter Summary

פרק זה הציג שיטה מערכתית לתכנון פרומפטים. שלושה עקרונות מרכזיים:

1. **עקרון האטום:** פרק משימות מורכבות לפרומפטים פשוטים. כל פרומפט עושה דבר אחד. הרכב אותם יחד לפתרון שלם.

2. **השוואה מדעית:** השווה פרומפטים באופן מבוקר. שנה משתנה אחד בכל פעם. מדוד את ההשפעה.

3. **כלל השלוש:** כתוב שלוש גרסאות. בדוק את כולן. בחר את המתאימה ביותר למשימה ולמודל.

בפרק הבא נלמד על תבניות פרומפט מתקדמות: Chain-of-Thought, ReAct, ו-Tree-of-Thoughts. תבניות אלה מרחיבות את הכלים שלנו ליצירת פרומפטים יעילים יותר.

### 3 תבניות פרומפט מתקדמות וארכיטקטורות

#### מטרות הלמידה

בסיום פרק זה, הקורא יוכל:

- לנתח לעומק את Chain-of-Thought, ReAct, ו-Tree-of-Thoughts
- ליישם few-shot learning אופטימלי
- לבנות מערכות role-based prompting
- להבין את היתרונות והחסרונות של כל תבנית

#### 3.1 מבוא: תפקיד המתכנת בעידן ההסקה

מודל שפה גדול הוא יצור מוזר. הוא יודע הכל, אבל לא יודע לחשוב. נסביר. כשאתה שואל אותו שאלה, הוא קופץ לתשובה. הוא לא עוצר לחשוב. הוא לא שואל את עצמו: "האם אני בטוח?". הוא פשוט עונה. לפעמים הוא צודק. לפעמים הוא טועה. ולפעמים הוא ממציא דברים שלא היו ולא נבראו. החוקרים קוראים לזה "הזיות" (Hallucinations).

אבל הנה הבעיה האמיתית: אתה לא יכול לתקן את זה. לא באמת. אתה לא יכול לפתוח את המודל ולשנות אותו. אתה לא יכול לאמן אותו מחדש. זה עולה מיליונים. זה לוקח חודשים. אתה עובד ברמת ההסקה (Inference) בלבד. המודל הוא קופסה שחורה. אתה יכול רק לדבר איתו. וכאן נכנס המתכנת לתמונה.

##### 3.1.1 הפיגום החיצוני

אם אתה לא יכול לשנות את המוח – שנה את הסביבה. חשוב על ילד עם הפרעת קשב. הוא לא טיפש. הוא פשוט קופץ ממחשבה למחשבה. מה עושים? נותנים לו מבנה. רשימת משימות. שלב אחרי שלב. אותו דבר עם מודלי שפה. הטכניקות שנלמד בפרק זה – Chain-of-Thought, ReAct, Tree-of-Thoughts – עושות בדיוק את זה. הן בונות **פיגום חיצוני** (External Scaffolding) סביב המודל. הפיגום מכתוב לו איך לחשוב. הוא לא משנה את המודל. הוא משנה את השיחה [8].

##### 3.1.2 שני מישורים של בקרה

יש שתי דרכים לבנות פיגום.

**הדרך הראשונה: הנחיה מובנית** (Structured Prompting). אתה כותב בפרומפט: "חשוב צעד אחר צעד". "הסבר את ההיגיון שלך". "אל תקפוץ למסקנה". המודל מקשיב. הוא מאט. הוא מראה את העבודה. זה Chain-of-Thought. זה עובד. זה פשוט.

**הדרך השנייה: לולאות בקרה חיצוניות.** לפעמים הנחיה לא מספיקה. לפעמים צריך קוד. ב-ReAct, המודל צריך לגשת לכלים חיצוניים. מנוע חיפוש. מחשבון. מאגר נתונים. מי מנהל את זה? לא המודל. אתה. ב-Tree-of-Thoughts, המודל צריך לחקור מספר נתיבים במקביל. מי מחליט איזה נתיב לגזור? לא המודל. אתה. אתה כותב קוד שמריץ את המודל. שולח לו שאלות. מקבל תשובות. מעריך אותן. מחליט מה לעשות הלאה. הקוד שלך הוא הפיגוס. הוא מנהל את המצב. הוא מתאם את התהליך [14].

### 3.1.3 המסקנה החשובה

המודל לא יוזם את הטכניקות האלה בעצמו. זה לא שהוא לא יכול. זה שהוא לא יודע שהוא צריך. הוא אומן על טקסטים. הוא למד לחזות את המילה הבאה. אף אחד לא לימד אותו לעצור ולחשוב. **זה התפקיד שלך.** אתה המתכנת. אתה מגדיר את הפונקציה של הפרומפט. אתה קובע את הגבולות. אתה בונה את הפיגוס שהופך מודל שקופץ למסקנות – למודל שחושב. בפרק זה נלמד שלוש תבניות מרכזיות:

1. Chain-of-Thought – חשיבה צעד אחר צעד, באמצעות הנחיה מובנית
2. ReAct – שילוב חשיבה עם פעולה, באמצעות לולאות בקרה חיצוניות
3. Tree-of-Thoughts – חקירת מספר נתיבים, באמצעות קוד שמנהל את העץ

## 3.2 Chain-of-Thought: שרשרת מחשבה

### 3.2.1 הגדרה פורמלית

#### הגדרה: ]

Chain-of-Thought (CoT) טכניקת הנדסת פרומפטים המאפשרת למודלי שפה גדולים (LLMs) להתמודד עם משימות חשיבה מורכבות. השיטה יוצרת רצף ליניארי של שלבי ביניים לוגיים ("מחשבות"). השלבים מקשרים בין הקלט לבין הפלט הסופי, במקום לחזות את התשובה באופן ישיר.

### 3.2.2 הסבר נרטיבי

איך אנחנו פותרים בעיה קשה? אנחנו לא קופצים לתשובה. אנחנו חושבים בשלבים. קח ילד שלומד חשבון. הוא לא כותב רק "התשובה היא 42". הוא כותב: "קודם נחבר את האחדות. אחר כך נעבור לעשרות". השלבים האלה הם לא סתם נוחות. הם הדרך שבה המוח עובד.

מודלי שפה למדו מטקסטים של בני אדם. הם ראו מיליארדי דוגמאות של תשובות. אבל רוב התשובות האלה הן "קפיצה" ישירה למסקנה. בלי הסבר. בלי שלבים. כשמבקשים מהמודל "לחשוב בקול רם" – הכל משתנה [8].

### 3.2.3 דוגמה פשוטה

✓ **עשה:** עם Chain-of-Thought

**שאלה:** "לדני 5 תפוחים. הוא זרק 2 ומצא 3. כמה יש לו?"

**תשובת המודל:**

בהתחלה היו 5 תפוחים.

הוא זרק 2, אז נשארו 3.

הוא מצא 3, אז 3 ועוד 3 זה 6.

**התשובה הסופית:** 6 תפוחים.

▲ **אל תעשה:** בלי Chain-of-Thought

**שאלה:** "לדני 5 תפוחים. הוא זרק 2 ומצא 3. כמה יש לו?"

**תשובת המודל:** "6".

(המודל נותן תשובה נכונה, אבל אין דרך לבדוק את ההיגיון)

### 3.2.4 גרסה מתקדמת: Chain-of-Thought++

**Chain-of-Thought++**

CoT עם self-consistency voting: קיימים 3 נתיבי חשיבה, הצבעת רוב. ב-2025 הוכח ש-5 נתיבים עם temperature שונה מוסיפים 12% ל-accuracy.

### 3.2.5 מהות השיטה המתקדמת

בני אדם לא קופצים למסקנות. הם חושבים בשלבים. כשאנחנו פותרים בעיה מתמטית, אנחנו כותבים את השלבים. כשאנחנו מנתחים טקסט, אנחנו מפרקים אותו לחלקים. Chain-of-Thought מלמד מודלים לעשות את אותו הדבר [8].

השיטה פשוטה. במקום לבקש תשובה ישירה, מבקשים מהמודל להסביר את החשיבה. במקום "מה התשובה?", שואלים "חשוב צעד אחר צעד, ואז תן תשובה". ההבדל הזה משנה הכל.

Wei ועמיתיו ב-Google הראו תוצאות מדהימות [8]. על מבחן GSM8K של בעיות מתמטיות, הדיוק עלה מ-17.9% ל-58.1%. זה שיפור של יותר מפי שלושה.

### 3.2.6 הצבעת עקביות עצמית

אבל יש בעיה. לפעמים המודל טועה בשלב אחד. הטעות גוררת טעויות נוספות. הפתרון: ליצר מספר נתיבי חשיבה ולהצביע על התשובה הנפוצה ביותר. Wang ועמיתיו קראו לשיטה Self-Consistency [10]. הרעיון פשוט. מריצים את אותו פרומפט 5 פעמים עם temperature שונה. כל הרצה מייצרת נתיב חשיבה שונה. בסוף, בוחרים בתשובה שהופיעה הכי הרבה. התוצאות מרשימות. על GSM8K, השיטה הוסיפה 17.9% לדיוק. על SVAMP, ההוספה הייתה 11%. על AQuA, 12.2%. היתרון: השיטה לא דורשת אימון נוסף. היא עובדת "מהקופסה" [10].

משוואה 8 מתארת את החישוב המתמטי:

#### משוואה 1.3: הצבעת Self-Consistency ב-CoT

$$(8) \quad \hat{y} = \arg \max_{y \in Y} \sum_{i=1}^N \mathbb{1}[y_i = y], \quad y_i \sim P(y|x, \tau_i)$$

במשוואה זו,  $N$  הוא מספר הנתיבים.  $\tau_i$  הוא ה-temperature של כל נתיב.  $\mathbb{1}[y_i = y]$  היא פונקציית אינדיקטור שסופרת כמה פעמים הופיעה כל תשובה.

### 3.3 ReAct: שילוב חשיבה ופעולה

#### 3.3.1 הגדרה פורמלית

##### הגדרה: ]

[ReAct (Reasoning + Acting) פרדיגמה המשלבת יכולות הסקה (Reasoning) עם ביצוע פעולות (Acting). השיטה מאפשרת למודל שפה ליצור עקבות חשיבה מילוליות ולבצע פעולות מול כלים חיצוניים. הכלים יכולים להיות מנועי חיפוש, מחשבוני, או מאגרי מידע. התהליך הוא משולב, אינטראקטיבי ומחזורי – עד להשלמת המשימה.

#### 3.3.2 הסבר נרטיבי

מודל שפה הוא כמו מוח בצנצנת. הוא יודע הרבה, אבל הוא מנותק מהעולם. חשוב על זה כך: אם תשאל אותי "מה מזג האוויר בפרז עכשיו?" – אני לא יודע. אני יכול לנחש, אבל אין לי גישה למידע עדכני. אני צריך לחפש. ReAct נותן למודל את היכולת הזו [14].

השיטה עובדת במעגל פשוט:

1. **מחשבה** (Thought): "מה אני צריך לדעת?"

2. **פעולה** (Action): שימוש בכלי חיצוני להשגת המידע

### 3. תצפית (Observation): קריאת התוצאה שחזרה

המעגל חוזר על עצמו. המודל חושב, פועל, צופה, וחוזר לחשוב. זה הופך אותו מ"מוח בצנצנת" לסוכן פעיל בעולם.

#### 3.3.3 דוגמה פשוטה

✓ עשה: ReAct בפעולה

שאלה: "האם יורד גשם בעיר הבירה של צרפת?"

מחשבה: אני צריך לדעת מה הבירה של צרפת, ואז לבדוק מזג אוויר.

פעולה 1: `search("capital of France")`

תצפית: פריז.

פעולה 2: `search("weather Paris now")`

תצפית: שמש, 25 מעלות.

תשובה סופית: לא, בפריז כרגע שמש.

#### 3.3.4 גרסה מתקדמת: ReAct 2.0

##### ReAct 2.0

שינוי מ-Think-Act-Observe ל-Act-Observe-Reflect-Adapt. שלב ה-Reflect מוסיף שכבה מטא-קוגניטיבית.

#### 3.3.5 מעבר לפרדיגמה חדשה

חשיבה לבדה לא מספיקה. לפעמים צריך לפעול בעולם. ReAct משלב את שניהם: חשיבה (Reasoning) ופעולה (Acting) [14].

המודל חושב. הוא מחליט על פעולה. הוא מבצע אותה. הוא צופה בתוצאה. ואז הוא חוזר לחשוב. הלולאה הזו ממשיכה עד שהמשימה הושלמה.

Yao ועמיתיו מ-Princeton ו-Google פרסמו את השיטה ב-ICLR 2023 [14]. על HotpotQA, השיטה הפחיתה הזיות. על ALFWorld, היא שיפרה את ההצלחה ב-34%.

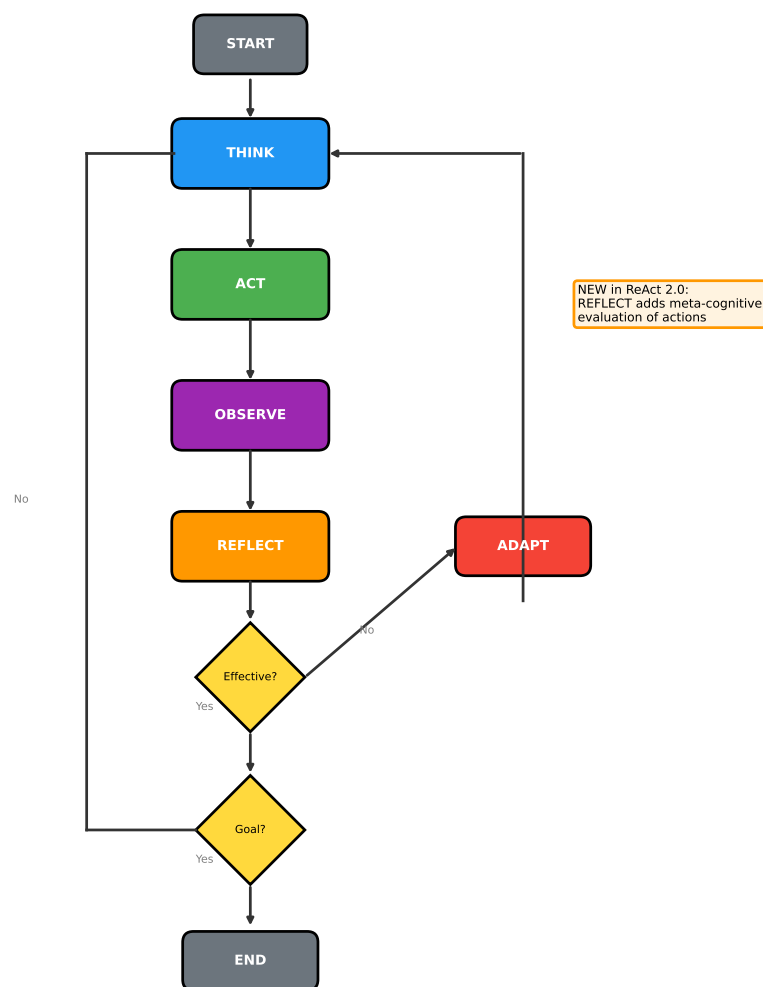
#### 3.3.6 ארכיטקטורת הזרימה

ב-ReAct המקורי היו שלושה שלבים: חשב, פעל, צפה (Think-Act-Observe). ב-ReAct 2.0, נוסף שלב רביעי: רפלקציה (Reflect).

השלב החדש שואל: "האם הפעולה הייתה אפקטיבית?". אם כן, ממשיכים. אם לא, משנים אסטרטגיה (Adapt). זוהי שכבה מטא-קוגניטיבית. המודל לומד לחשוב על החשיבה שלו.

איור 6 מציג את הארכיטקטורה המלאה.

**Figure 3.2: ReAct 2.0 Architecture**  
**Think-Act-Observe-Reflect-Adapt Loop**



איור 6: ארכיטקטורת ReAct 2.0: מעגל Act-Observe-Reflect-Adapt



## 3.4 Tree-of-Thoughts: עץ מחשבות

### 3.4.1 הגדרה פורמלית

#### הגדרה: ]

Tree-of-Thoughts (ToT) מסגרת אלגוריתמית המכלילה את גישת Chain-of-Thought. השיטה מאפשרת למודל לחקור מספר נתיבי חשיבה אפשריים במקביל ("ענפים"). המודל מבצע הערכה עצמית (Self-Evaluation) לכל צומת בעץ. הוא מחליט האם להמשיך בנתיב מסוים, לחזור אחורה (Backtrack), או לפצל את החשיבה לכיוונים חדשים.

### 3.4.2 הסבר נרטיבי

אם Chain-of-Thought הוא קו ישר, Tree-of-Thoughts הוא מבוך. חשוב על שחמטאי. הוא לא חושב רק על המהלך הבא. הוא חושב על עשרה מהלכים קדימה. יותר מזה – הוא בוחן כמה אפשרויות במקביל. "אם אני אעשה כך, מה יקרה? ואם אעשה אחרת?" Tree-of-Thoughts נותן למודל את היכולת הזו [15]. המודל בונה עץ החלטות. בכל צומת יש מחשבה – פתרון חלקי. הוא נותן ציון לכל כיוון: "האם הכיוון הזה מבטיח?" אם הוא מגיע למבוי סתום, הוא יודע לחזור אחורה ולנסות כיוון אחר.

### 3.4.3 דוגמה פשוטה

✓ **עשה:** Tree-of-Thoughts בפעולה  
**משימה:** "כתוב משפט סיום למותחן."

#### שלב 1 – יצירת ענפים:

המודל מציע 3 רעיונות:

(א) הרוצח הוא השוטר

(ב) הכל היה חלום

(ג) החייזרים פלשו

#### שלב 2 – הערכה:

(ב) קלישאתי → ציון נמוך

(ג) לא קשור לז'אנר → ציון נמוך

(א) מעניין ומפתיע → ציון גבוה

#### שלב 3 – המשך:

המודל בוחר ב-(א) וממשיך לפתח רק אותו.

### 3.4.4 גרסה מתקדמת: Tree-of-Thoughts עם תקציב כפוי

## Budget Forcing עם Tree-of-Thoughts

הגבלת מספר הצמתים ל- $B = 2^{\text{depth}}$  עם pruning דינמי.

### 3.4.5 חקירה מבנית

לפעמים יש יותר מדרך אחת לפתור בעיה. Chain-of-Thought בוחר דרך אחת. Tree-of-Thoughts חוקר מספר דרכים במקביל [15].  
Yao ועמיתיו מ-Princeton הציגו את השיטה ב-NeurIPS 2023 [15]. הרעיון: לארגן את החשיבה כעץ. כל צומת הוא "מחשבה" – פתרון חלקי. המודל מעריך כל מחשבה ומחליט אם להמשיך או לגזום.  
התוצאות מרשימות. על משחק Game of 24, GPT-4 עם CoT פתר 4% בלבד. עם Tree-of-Thoughts, ההצלחה עלתה ל-74%. זו קפיצה של פי 18.

### 3.4.6 גיזום דינמי

עץ יכול לגדול ללא גבולות. זה בזבוז משאבים. לכן משתמשים בגיזום (pruning). הרעיון: לחתוך ענפים עם ציון נמוך. להתמקד בענפים מבטיחים.  
משוואה 9 מגדירה את תנאי הגיזום:

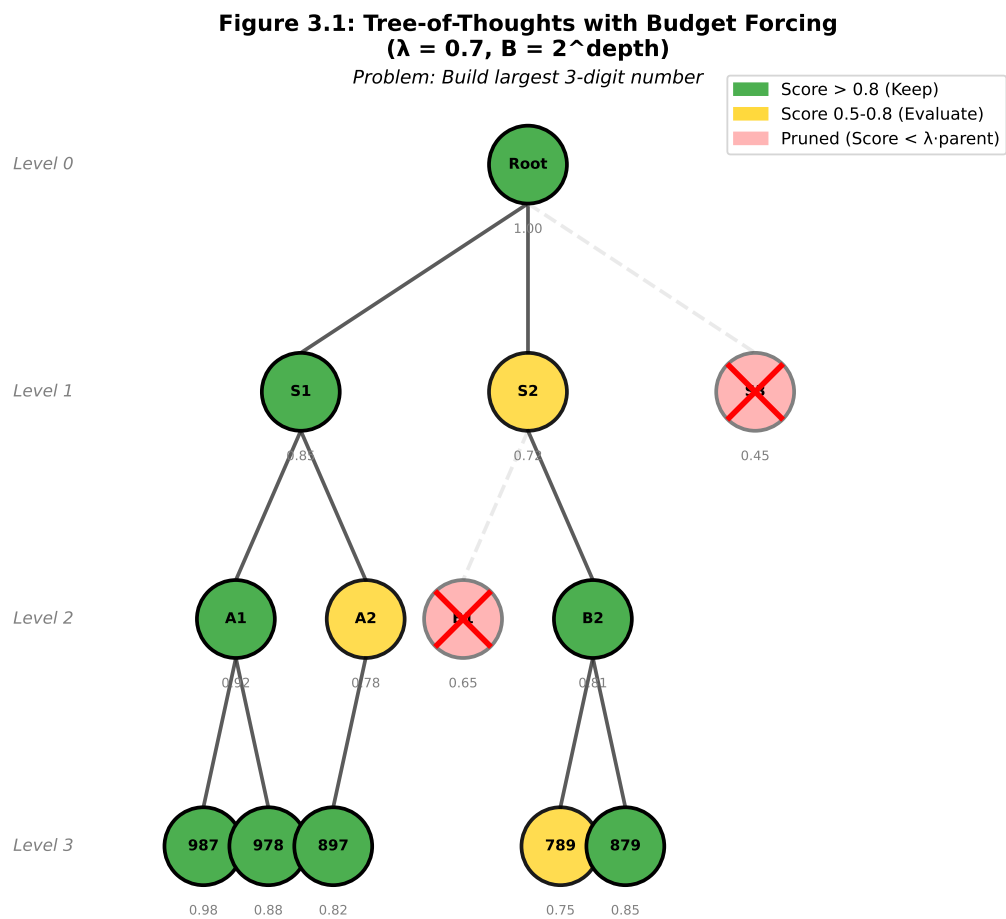
#### משוואה 2.3: תנאי Pruning ב-Tree-of-Thoughts

$$(9) \quad \text{Prune}(n) = \begin{cases} \text{True} & \text{if } \frac{\sum_{i=1}^k \text{score}(c_i)}{k} < \lambda \cdot \text{score}(n) \\ \text{False} & \text{otherwise} \end{cases}$$

במשוואה זו,  $n$  הוא הצומת הנבדק.  $c_i$  הם הילדים שלו.  $\lambda = 0.7$  הוא סף האיכות. אם ממוצע ציוני הילדים נמוך מ-70% מציון ההורה, גוזמים.  
איור 7 מציג דוגמה לעץ עם גיזום.

## 3.5 הנחיית תפקידים: Role-Based Prompting

האם למודל יש "אישיות"? לא באמת. אבל אפשר לתת לו תפקיד. "אתה פרופסור לכלכלה". "אתה מתכנת מומחה". "אתה מורה לילדים". התפקיד משפיע על התשובות.  
Kong ועמיתיו חקרו את ההשפעה [16]. על מבחן AQUA, הדיוק עלה מ-53% ל-63%. אבל לא תמיד יש שיפור. Zheng ועמיתיו הראו שבחלק מהמקרים אין הבדל [17].  
המסקנה: התפקיד עוזר כשהוא ספציפי ורלוונטי למשימה. "אתה מומחה" פחות טוב מ"אתה מומחה לכלכלה התנהגותית שמלמד סטודנטים".  
משוואה 01 מתארת את ההשפעה:



איור 7: עץ מחשבות עם גיזום: ענפים עם ציון נמוך נגזמים

### משוואה 3.3: השפעת Role-Based Prompting

$$(10) \quad \Delta_{\text{role}} = \mathbb{E}[\text{Acc}_{\text{with role}}] - \mathbb{E}[\text{Acc}_{\text{no role}}] = \beta \cdot \text{RoleSpecificity} + \epsilon$$

במשוואה זו,  $\beta$  מודד את עוצמת ההשפעה. מחקרים מראים  $\beta = 0.12$  למשימות חשיבה. למשימות יצירה, ההשפעה חלשה יותר:  $\beta = 0.06$  [18].

### 3.6 למידה מדוגמאות: אופטימיזציה של Few-Shot

כמה דוגמאות צריך? זו שאלה חשובה. יותר מדי – בזבוז. פחות מדי – חוסר דיוק. Brown ועמיתיו ב-OpenAI הראו את הכוח של למידה מדוגמאות [19]. GPT-3 עם 175 מיליארד פרמטרים למד מ-10-100 דוגמאות בלבד. לא היה צורך באימון נוסף. הדוגמאות הספיקו.

טבלה 5 מציגה את ההשפעה של מספר הדוגמאות:

טבלה 5: אופטימיזציית few-shot learning

מספר דוגמאות	תקציב טוקנים	שיפור דיוק	השפעת סדר
1	+150	+5%	לא רלוונטי
3	+450	+11%	השפעה גבוהה
5	+750	+14%	השפעה בינונית
8	+1200	+15%	רעש

המסקנה: 3 דוגמאות הן נקודת האיזון האופטימלית. יותר מכך – תשואה פוחתת.

### 3.7 השוואה מערכתית בין תבניות

איזו תבנית לבחור? התשובה תלויה במשימה.

טבלה 6 משווה את התבניות השונות:

הכלל: התחל פשוט. Zero-shot לא עובד? נסה Few-shot. עדיין לא מספיק? עבור ל-CoT. צריך לגשת למידע חיצוני? השתמש ב-ReAct. בעיה מורכבת עם מספר פתרונות? נסה ToT.

### 3.8 דוגמאות עשה ואל תעשה: Do and Don't Examples

✓ עשה: ReAct Structured  
"שאלה: מה המטבע של צ'כיה?"

טבלה 6: השוואת תבניות פרומפטים לפי סיבוכיות

תבנית	עומס טוקנים	שיפור דיוק	מכפיל זמן	מתאים למשימות
Zero-shot	1.0x	בסיס	1.0x	פשוטות
Few-shot	1.3x	+8-12%	1.0x	דורשות דוגמאות
CoT	1.8x	+15-20%	1.5x	חשיבה, חשבון
ReAct	2.5x	+18-25%	2.0-3.0x	כלים, חיפוש
ToT	4.0x	+22-30%	4.0-6.0x	יצירתיות

מחשבה: צ'כיה היא באירופה, אני צריך לבדוק.  
פעולה: `search(query='Czech Republic currency 2025')`  
תצפית: התוצאה מראה שצ'כיה משתמשת בקורונה צ'כית.  
תשובה: קורונה צ'כית."

▲ **אל תעשה:** ReAct Unstructured  
"מה המטבע של צ'כיה? תחפש ותגיד לי."

✓ **עשה:** Tree-of-Thoughts  
"בעיה: מצא את הדרך הקצרה ביותר בגרף.  
רמת 1: 3 אסטרטגיות (A\*, DFS, BFS).  
רמת 2: כל אסטרטגיה מייצרת 2 פתרונות.  
בחר את הפתרון עם הציון הטוב ביותר."

▲ **אל תעשה:** Single Path  
"מצא את הדרך הקצרה ביותר."

✓ **עשה:** Role-Based Specific  
"אתה פרופסור לכלכלה התנהגותית. הסבר את פרדוקס הבחירה לסטודנטים ב-200 מילים. השתמש בדוגמה מעולם הקנייה המקוונת."

### 3.9 חלון ההקשר בפעולה: Context Window Mechanics

מודל שפה לא זוכר. הוא קורא מחדש. זו נקודה קריטית. בכל פעם שאתה שולח הודעה, המודל מקבל את כל ההיסטוריה מחדש. הוא לא "נזכר" במה שאמר קודם. הוא פשוט רואה את זה בחלון הכניסה. ככל שהשיחה מתארכת, החלון מתמלא. ובסוף – הוא נגמר. בחלק זה נראה איך כל שיטה משפיעה על חלון ההקשר.

#### 3.9.1 Chain-of-Thought: צמיחת החלון

ב-CoT, המודל מייצר "טוקנים של חשיבה" שנכנסים לחלון. המשפט "בוא נחשוב צעד אחר צעד" משנה את התנהגות המודל. במקום לקפוץ לתשובה, הוא מייצר שלבי ביניים. השלבים האלה תופסים מקום.

##### שלב 1: השאלה הראשונה ( $T_1$ )

טבלה 7: חלון הקשר ב-CoT – שלב 1

חלון כניסה	חלון יציאה
<p><b>תכרעם תייחנה:</b> ןייטצמ תודיח רתופ התא</p> <p><b>שמתשמ:</b> הנוק אוה. סינט ירודכ 5 שי ר'גורל סירודכ 3 תחא לכבש תואספוק 2 ול שי המכ 2. דבאמ אוה</p> <p><b>רגירט:</b> דעצ רחא דעצ הז לע בושחנ אוב</p>	<p><b>לדומה תבושת</b> דעצ רחא דעצ בשחנ אוב סירודכ 5: הלחתה 1. <math>6 = 3 \times 2</math> תואספוק 2: היינק 2. סירודכ סירודכ 11: <math>11 = 5 + 6</math> לכה דס 3. סירודכ 9: <math>9 = 11 - 2</math> דוביא 4. <b>הבושתה 9</b></p>

##### שלב 2: שאלת המשך ( $T_2$ )

שים לב: חלון הכניסה עכשיו מכיל את כל מה שקרה ב- $T_1$ .

טבלה 8: חלון הקשר ב-CoT – שלב 2

חלון כניסה	חלון יציאה
<p><b>תכרעם תייחנה</b> [ל"נכ] ...סירודכ 5 שי ר'גורל <math>T_1</math> שמתשמ 9: הבושתה ...בשחנ אוב <math>T_1</math> לדומ</p> <p><b><math>T_2</math> שמתשמ:</b> הווש סירודכה תא קלחי ר'גור סא דחא לכ המכ, סידלי 3 ןיב הוושב לבקי</p>	<p><b>לדומה תבושת</b> שי, תמדוקה הבושתה לע ססבתהב סירודכ 9 בושיח <math>3 = 9 \div 3</math> <b>סירודכ 3 לבקי דלי לכ</b></p>

### 3.9.2 ReAct: עצירות וחזרות

ב-ReAct, החלון גדל בקפיצות. המודל עוצר, מבצע פעולה, ומקבל תוצאה. כל מחזור מוסיף שלושה חלקים: Observation, Action, Thought.

טבלה 9: חלון הקשר ב-ReAct – מחזור שלם

חלון כניסה	חלון יציאה
<p><b>תכרעם תייחנה:</b> שופיח עונמל השיג דל שי טמרופב שמתשה Thought: [הבשחמ] Action: [הלועפ] Observation: [האצות]</p> <p><b>שמתשמ:</b> היה המו Super Bowl 2024 ב- הכז ימ שרפהה?</p> <p><b>(תכרעמהמ) תיפצת:</b> 25-22. וחצינ ספ'צ יטיס סזנק</p>	<p><b>(1 בלש) לדומ:</b> הכז ימ אוצמל ד'רצ ינא Thought: Super Bowl 2024 ב- Action: Search[“Super Bowl 2024 winner”] [ואכ תרצוע תכרעמה] <b>(2 בלש) לדומ:</b> שרפהה תא בשחא וישכע Thought: 22 תוחפ 25 Answer: 3 שרפהה. וכז ספ'צה תודוקנ</p>

שים לב: ה-Observation לא נוצר על ידי המודל. הוא מוזרק על ידי המערכת החיצונית. זה מה שהופך את ReAct לתלוי בקוד חיצוני.

### 3.9.3 Tree-of-Thoughts: ענפים מקבילים

ב-ToT, החלון מתפצל. המודל מייצר מספר אפשרויות, מעריך כל אחת, ובוחר. ביישום מתקדם, זה דורש **מספר קריאות** למודל – אחת לכל ענף.

טבלה 10: חלון הקשר ב-ToT – הערכת ענפים

חלון כניסה	חלון יציאה
<p><b>תכרעם תייחנה:</b> החמומ רפוס התא הלחתהל סינוש סינוויכ 3 עצה 1-10. וויצב דחא לכ ד'רעה רתויב חיטבמה תא רחב</p> <p><b>שמתשמ:</b> שלב לע רופיסל החיתפ טפשם בותכ ןמזה תא רצועש וועש אצומש</p>	<p><b>לדומ:</b> וועשה תא סירה שלבה 1 וויכ אפק סלועהו ידמ ילאנב – 6/10 Score: ”חצנל 12:00 התייה העשה” 2 וויכ שלבה למלמ יתריוואו ורקסמ – 9/10 Score: סושו רותפכה לע ץחל אוה 3 וויכ הרק אל רבד יתאשילק – 7/10 Score: 2 וויכ סע דישממ <b>הטלחה</b></p>

### 3.9.4 מה קורה כשהחלון נגמר?

זו שאלה קריטית. לכל מודל יש מגבלת Context Window:

- GPT-4: עד 128000 טוקנים

- Claude 3: עד 200000 טוקנים

- Gemini 1.5: עד 1000000 טוקנים

כשהחלון מתמלא, המערכת חייבת **למחוק** חלק מההיסטוריה. בדרך כלל, המידע הישן ביותר נמחק ראשון (FIFO). המשמעות: בשיחה ארוכה, המודל "ישכח" את תחילת השיחה. הנחיית המערכת בדרך כלל נשמרת, אבל ההודעות הראשונות – יעלמו.

#### Context Management

ניהול חלון ההקשר הוא חלק קריטי מהנדסת פרומפטים. ככל שהשיטה מורכבת יותר (CoT < ReAct < ToT), כך היא צורכת יותר טוקנים. תכנון נכון מאזן בין עומק החשיבה לבין הזיכרון הזמין.

### 3.10 סיכום הפרק: Chapter Summary

למדנו שלוש תבניות מתקדמות להנדסת פרומפטים:

1. **Chain-of-Thought**: מלמד מודלים לחשוב בשלבים. Self-Consistency מוסיפה הצבעה בין נתיבים.

2. **ReAct**: משלב חשיבה עם פעולה. גרסה 2.0 מוסיפה רפלקציה.

3. **Tree-of-Thoughts**: חוקר מספר נתיבים במקביל. גיזום חוסך משאבים.

בנוסף, למדנו על:

- Few-shot learning – 3 דוגמאות הן נקודת האיזון

- Role-based prompting – תפקיד ספציפי עוזר יותר מתפקיד כללי

בפרק הבא נלמד כיצד להשתמש בתבניות אלו לבניית סוכני AI.



### מטרות הלמידה

בסיום פרק זה, הקורא יוכל:

- לעצב פרומפטים לסוכנים אוטונומיים
- לשלב state management ו- tool-use
- לבנות guardrails ומנגנוני בטיחות
- להבין את עקרונות ה-Agent-Computer Interface

### 4.1 מבוא: מהמענה לפעולה

מאז ומתמיד חלם האדם על עוזרים אוטונומיים. מהגולם של פראג ועד רובוטי המדע הבדיוני, החזון היה אחד: יצור שמבין הוראות ופועל לפיהן. כיום, חזון זה הופך למציאות. מודלי שפה גדולים (LLMs) כבר לא רק עונים על שאלות. הם יכולים לפעול בעולם האמיתי [14].

סוכני AI הם מערכות שמשלבות חשיבה עם פעולה. הם מקבלים משימה, מתכננים צעדים, ומבצעים אותם. לפעמים הם קוראים לכלים חיצוניים. לפעמים הם מחפשים מידע באינטרנט. תמיד הם צריכים לדעת מתי לעצור. אבל יש בעיה מהותית. מודלי שפה לא נבנו לפעולה. הם נבנו לחיזוי המילה הבאה. כדי להפוך אותם לסוכנים, צריך פרומפטים מיוחדים. פרומפטים שמגדירים לא רק מה לחשוב, אלא גם מה לעשות [20].

### 4.2 ממשק סוכן-מחשב: Agent-Computer Interface (ACI)

#### efafretnI retupmoC-tnegA

פרומפטים לסוכנים חייבים לכלול ארבעה רכיבים:

1. תיאורי כלים עם דוגמאות
2. פורמט מצב (state)
3. פרוטוקול טיפול בשגיאות
4. תנאי סיום

כמו שמחשב צריך ממשק משתמש, סוכן צריך ממשק לעולם. זהו ה-Agent-Computer Interface או ACI. ללא ממשק ברור, הסוכן לא יודע מה הכלים שלו. הוא לא יודע מה המצב

הנוכחי. הוא לא יודע מתי לעצור.  
מחקר Anthropic מ-2024 מראה תובנה מפתיעה [20]. הפתרונות הפשוטים עובדים הכי טוב. ארכיטקטורות מורכבות נכשלות. תבניות פשוטות וניתנות להרכבה מצליחות.

### 4.2.1 תבנית ReAct

תבנית ReAct היא הבסיס לרוב הסוכנים [14]. השם מגיע מ-Reasoning + Acting. הסוכן מבצע מעגל פשוט:

1. **חשוב** (Think): נתח את המצב

2. **פעל** (Act): בחר וקרא לכלי

3. **צפה** (Observe): קרא את התוצאה

4. חזור לשלב 1 עד להשגת המטרה

ב-2023 הוסיפו שלב רביעי: **הרהר** (Reflect). הסוכן בודק אם הפעולה הייתה אפקטיבית. אם לא, הוא משנה את האסטרטגיה. גרסה זו נקראת ReAct 2.0 [21].

### 4.3 ארכיטקטורות סוכנים: Agent Architectures

טבלה 11 מציגה את הארכיטקטורות הנפוצות. כל אחת מתאימה לסוג אחר של משימות.

טבלה 11: ארכיטקטורות פרומפטס לסוכנים

תבנית	רכיבים	Token Budget	Success Rate	מתאים ל...
ReAct Classic	Think, Act, Obs	300-500	0.78	שימוש בכלים פשוט
ReAct 2.0	+Reflect, Adapt	500-800	0.85	שימוש בכלים מורכב
Manager-Worker	+ מנהל עובדים	$400+n*200$	0.88	ריבוי סוכנים
Decentralized	+ מיון מומחים	$300+n*150$	0.82	ניתוב דינמי

כפי שרואים בטבלה, תבנית Manager-Worker משיגה את שיעור ההצלחה הגבוה ביותר. אבל היא גם הכי יקרה במונחי tokens. הבחירה תלויה במשימה ובתקציב.

## 4.4 מעקות בטיחות כפרומפטים: Guardrails as Prompts

stpmorP sa sliardrauG

כל guardrail הוא פרומפט בפני עצמו שרץ במקביל לסוכן. מבנה: "IF action violates policy THEN raise Exception".

סוכנים יכולים לטעות. הם יכולים לחשוף מידע רגיש. הם יכולים לבצע פעולות מסוכנות. לכן צריך מעקות בטיחות. NVIDIA פיתחה את NeMo Guardrails [22]. זו ספרייה שמוסיפה שכבות הגנה לסוכנים. כל שכבה היא פרומפט עצמאי. הפרומפטים רצים במקביל לסוכן. ישנם ארבעה סוגים עיקריים של guardrails:

1. סינון PII: מזהה מידע אישי רגיש

2. זיהוי Jailbreak: מזהה ניסיונות לעקוף הגבלות

3. בטיחות כלים: מדרג סיכון של כל פעולה

4. רלוונטיות: בודק אם התשובה קשורה לשאלה

טבלה 12 מפרטת את סוגי ה-guardrails ואת הפרומפטים שלהם.

טבלה 12: סוגי מעקות בטיחות ופרומפטים

סוג	דוגמת פרומפט	תנאי הפעלה	פעולה בהפרה
סינון PII	סרוק output עבור: ת.ז., כרטיס אשראי	התאמת regex	מיסוך + רישום
זיהוי Jailbreak	האם input-ה לחשוף system prompt?	prob > 0.7	חסימה התראה +
בטיחות כלים	דרג סיכון: read=1, write=2, delete=3	Risk > 2	אישור אנושי
רלוונטיות	האם output-ה קשור לשאלה?	similarity < 0.6	נסה שוב עם CoT

## 4.5 מטא-פרומפטינג: Meta-Prompting

פרומפט שכותב פרומפטים. ב-2024 הוכח ש-meta-prompt עם 2 דוגמאות מגדיל את שיעור ההצלחה של סוכנים ב-34% [23].

מה אם במקום לכתוב פרומפטים ידנית, נבקש מה-LLM לכתוב אותם? זו הרעיון מאחורי Meta-Prompting. פרומפט שמייצר פרומפטים. Kalai ו-Suzgun הראו ב-2024 שהגישה עובדת [23]. הם פיתחו מסגרת שנקראת Task-Agnostic Scaffolding. המסגרת מתמקדת במבנה המשימה, לא בתוכן. התוצאות היו מרשימות. מודל Qwen-72B עם meta-prompt בודד השיג תוצאות state-of-the-art. היתרונות של Meta-Prompting:

1. **יעילות tokens**: פחות tokens נדרשים

2. **השוואה הוגנת**: פחות תלות בדוגמאות ספציפיות

3. **יכולת zero-shot**: עובד ללא דוגמאות

#### 4.6 נוסחאות מתמטיות: Mathematical Formulas

הבנת הביצועים של סוכנים דורשת מודלים מתמטיים. שלוש משוואות מרכזיות מגדירות את ההצלחה של סוכן.

##### משוואה 1.4: שיעור הצלחת סוכן

$$(11) \quad \text{Success} = P(\text{correct action}) \cdot \prod_{i=1}^T P(\text{observation}_i \text{ valid}) \cdot P(\text{termination})$$

משוואה 11 מראה שהצלחה היא מכפלה של שלושה גורמים:

-  $P(\text{correct action})$ : ההסתברות לבחור פעולה נכונה

-  $\prod_{i=1}^T P(\text{observation}_i \text{ valid})$ : ההסתברות שכל התצפיות תקינות

-  $P(\text{termination})$ : ההסתברות לעצור בזמן הנכון

כאשר  $T$  הוא מספר הצעדים. שימו לב: אם סוכן עושה הרבה צעדים, ההסתברות הכוללת יורדת. לכן סוכנים טובים ממזערים את מספר הצעדים.

##### משוואה 2.4: יעילות gnitpmorP-ateM

$$(12) \quad \text{Eff}_{\text{meta}} = \frac{\text{Acc}_{\text{meta-generated}} - \text{Acc}_{\text{human-written}}}{\text{Acc}_{\text{human-written}}} \times 100\%$$

משוואה 21 מחשבת את השיפור שמביא Meta-Prompting. המחקרים מראים  $\text{Eff}_{\text{meta}} = 34\% \pm 5\%$  עבור סוכנים חדשים [23].

### משוואה 3.4: ביטחון בבחירת כלי

$$(13) \quad \text{Select}(t) = \frac{\exp(\text{sim}(q, d_t))}{\sum_{j=1}^M \exp(\text{sim}(q, d_j))} \cdot \mathbb{I}[\text{guardrail}(t) = \text{pass}]$$

משוואה 31 מגדירה איך סוכן בוחר כלי. המשוואה משלבת שני גורמים:

- **דמיון סמנטי:**  $\text{sim}(q, d_t)$  בין השאילתה  $q$  לתיאור הכלי  $d_t$

- **מעבר guardrail:**  $\mathbb{I}[\text{guardrail}(t) = \text{pass}]$

שימו לב: כלי שנחסם על ידי guardrail מקבל ציון אפס. זה מבטיח בטיחות גם כשהכלי נראה רלוונטי.

### 4.7 שימוש בכלים: Tool-Use

היכולת להשתמש בכלים היא מה שמבדיל סוכנים מ-chatbots. Toolformer הראה שמודלים יכולים ללמוד להשתמש בכלים בעצמם [24]. ToolLLM הרחיב את זה ל-16000 APIs [25]. Gorilla הראה איך לחבר מודלים ל-APIs מאסיביים [26]. OpenAI הציגה ב-2024 את Structured Outputs [27]. זה מבטיח שהפלט תואם לסכמה מוגדרת מראש. ב-evaluations מורכבים, המודל החדש השיג 100% דיוק. לעומת זאת, GPT-4 הישן השיג פחות מ-40%.

Berkeley Function-Calling Leaderboard מודד את היכולת הזו [28]. הוא בודק:

- קריאות מרובות לפונקציות

- קריאות מקבילות

- קריאות רב-שלביות

### 4.8 דוגמאות עשה ואל תעשה: Do and Don't Examples

הבדל קטן בפרומפט יכול לעשות הבדל גדול בביצועים. להלן דוגמאות מעשיות.

✓ **עשה:** Tool Specification

"כלי: search\_database

תיאור: מחפש במסד נתונים לקוחות.

פרמטרים: query (string, example: 'age > 30 AND city="TLV"'), limit (int, default=10).

מחזיר: List[Customer] עם שדות: ".id, name, age"

▲ **אל תעשה:** Vague Tool

"כלי: search. תשתמש בו לחיפוש."

ההבדל ברור: הפרומפט הטוב מספק דוגמאות ופרמטרים. הפרומפט הגרוע לא מסביר כלום.

✓ **עשה:** State Management

"State: {'step': 2, 'last\_action': 'search', 'results': [...], 'goal': 'find premium customers'}"  
בהתבסס על ה-state, בחר את הפעולה הבאה."

▲ **אל תעשה:** No State

"תמשיך לחפש."

✓ **עשה:** Guardrail as Prompt

"אם הפעולה היא 'delete\_user', בדוק: user.status == 'inactive' AND request.approval == True. אחרת, דחה את הבקשה."

▲ **אל תעשה:** No Guardrails

"תמחק משתמשים לפי הצורך."

✓ **עשה:** Meta-Prompting

"אתה מומחה לכתיבת פרומפטים. כתוב פרומפט ל-summarization של דוחות כספיים. הפרומפט חייב לכלול: (1) תפקיד, (2) פורמט, (3) מדדים חשובים. דוגמה: ..."

▲ **אל תעשה:** Direct Prompt

"כתוב פרומפט לסיכום דוחות."

## 4.9 סיכום הפרק: Chapter Summary

בפרק זה למדנו כיצד לעצב פרומפטים לסוכני AI. הנקודות המרכזיות הן:

1. **ממשק ACI:** כל סוכן צריך ארבעה רכיבים: כלים, מצב, טיפול בשגיאות, ותנאי סיום.
  2. **תבנית ReAct:** מעגל של חשיבה-פעולה-תצפית הוא הבסיס לרוב הסוכנים.
  3. **מעקות בטיחות:** כל guardrail הוא פרומפט עצמאי שרץ במקביל.
  4. **Meta-Prompting:** פרומפטים שכותבים פרומפטים יכולים לשפר ביצועים ב-34%.
  5. **שימוש בכלים:** תיאור מדויק של כלים קריטי להצלחה.
- בפרק הבא נלמד על הערכה, אופטימיזציה והטמעה בייצור.

## 5 הערכה, אופטימיזציה והטמעה בייצור

### מטרות הלמידה

בסיום פרק זה, הקורא יוכל:

- ליישם מתודולוגיית A/B testing לפרומפטים
- לנהל CI/CD version control לפרומפטים
- לבצע מעקב (monitoring) וזיהוי drift
- להבין את עקרונות האופטימיזציה לעלות-יעילות

### 5.1 מבוא: מהמעבדה לייצור

לפני הכתב, בני האדם סמכו על הזיכרון. כל סיפור שהועבר מפה לאוזן השתנה מעט. גרסה אחת הפכה לעשר. איש לא ידע מה היה המקור. הנדסת פרומפטים עוברת מהפכה דומה. מה שהתחיל כאומנות אישית הופך לתהליך תעשייתי. המעבר הזה דורש כלים חדשים: ניהול גרסאות, בדיקות מובהקות, וזיהוי סחיפה. ללא הכלים האלה, פרומפט שעובד היום עלול להיכשל מחר. מחקר של Booking.com בחן 150 מודלים מוצלחים [29]. הם גילו עובדה מפתיעה: שיפור בדיוק המודל לא תמיד מוביל לשיפור עסקי. הדרך היחידה לדעת אם שינוי עובד היא לבדוק אותו בייצור. זה מחייב ניסויים מבוקרים, לא תחושות בטן.

### 5.2 ניהול גרסאות פרומפטים: Prompt Versioning

gninoisreV tpmorP

כל פרומפט חייב לכלול:

1. Version ID סמנטי (X.Y.Z)

2. Test set hash – טביעת אצבע של נתוני הבדיקה

3. מדדי ביצוע: דיוק, זמן תגובה, עלות

4. תאריך הטמעה ושם הכותב

בתוכנה מסורתית, קוד הוא דטרמיניסטי. אותו קוד נותן אותה תוצאה. בפרומפטים, המצב שונה לחלוטין. מודל שפה אינו דטרמיניסטי. הוא יכול לתת תשובות שונות לאותו פרומפט.

לכן ניהול גרסאות בפרומפטים מורכב יותר. צריך לעקוב לא רק אחרי הטקסט, אלא גם אחרי הביצועים. כל גרסה חייבת לכלול מטא-דאטה מפורט. זה מאפשר לחזור אחורה כשמשוהו משתבש.

הגישה המומלצת היא Semantic Versioning [30]. גרסה ראשית (X) משתנה כשהפרומפט משתנה באופן משמעותי. גרסה משנית (Y) משתנה כשמוסיפים יכולות. גרסה קטינה (Z) משתנה כשמתקנים באגים.

### 5.3 מובהקות סטטיסטית: Statistical Significance

#### ecnacifingis lacitsitatS

לשינויים בפרומפט, דרוש גודל מדגם מינימלי. ל-95% ביטחון ו-80% עוצמה, נדרשים לפחות 1000 דגימות לכל קבוצה.

המהפכה המדעית לימדה אותנו דבר חשוב. תחושות בטן מטעות. רק ניסויים מבוקרים יכולים להבדיל בין אמת לאשליה. כשמשנים פרומפט, קל לחשוב שהוא השתפר. אבל ייתכן שהשיפור הוא מקרי. כדי לדעת בוודאות, צריך מספיק דגימות. הנוסחה לחישוב גודל המדגם היא:

#### משוואה 5.1: חישוב גודל מדגם

$$(14) \quad n = \frac{(z_{1-\alpha/2} + z_{1-\beta})^2 \cdot (\sigma_1^2 + \sigma_2^2)}{\Delta^2}$$

במשוואה הזו:

-  $z_{1-\alpha/2}$  הוא ערך קריטי לרמת המובהקות 1.96 ל-95%

-  $z_{1-\beta}$  הוא ערך קריטי לעוצמה 0.84 ל-80%

-  $\sigma$  הוא סטיית התקן של המדד

-  $\Delta$  הוא ההבדל המינימלי שרוצים לזהות

מחקר של Apple [31] הרחיב את הנוסחאות הללו. הם פיתחו שיטות לנתונים מתואמים. זה חשוב במיוחד כשמשתמשים חוזרים מספר פעמים.

### 5.4 מתודולוגיית בדיקות A/B: A/B Testing Methodology

הניסוי המבוקר האקראי היה המתנה הגדולה של הרפואה למדע. הוא אפשר להבדיל בין תרופות אמיתיות לפלצבו. בהנדסת פרומפטים, A/B testing ממלא תפקיד דומה. בניסוי A/B, מחלקים את המשתמשים לשתי קבוצות. קבוצת הביקורת מקבלת את הפרומפט הישן. קבוצת הטיפול מקבלת את הפרומפט החדש. משווים את הביצועים ובודקים אם ההבדל מובהק.



טבלה 13: מתודולוגיית בדיקות A/B לפרומפטים

וריאנט	משך	מדד הצלחה	p-value	החלטה
Control (v1.2)	7 ימים	0.891	–	בסיס
Treatment A	7 ימים	0.905	0.003	פרסום
Treatment B (CoT)	7 ימים	0.912	0.001	פרסום (מנצח)
Treatment C	7 ימים	0.898	0.089	דחייה

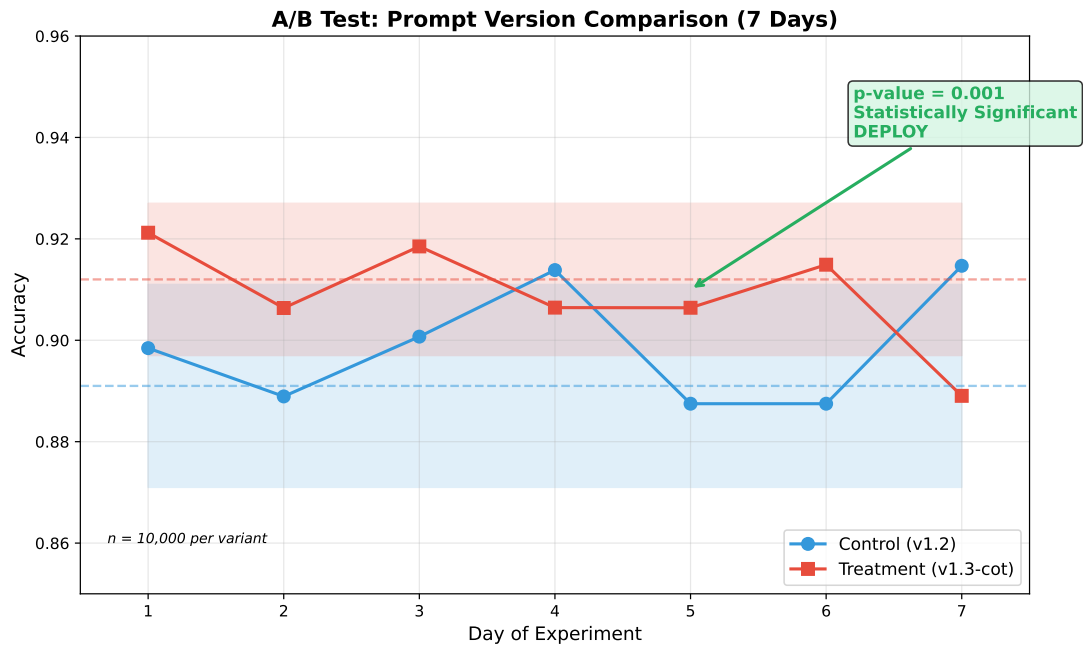
טבלה 13 מציגה דוגמה לניסוי עם ארבעה וריאנטים. רק וריאנטים עם p-value קטן מ-0.05 נחשבים מובהקים. וריאנט C נדחה למרות שהדיוק שלו גבוה מהבסיס. הסיבה: ההבדל לא מובהק סטטיסטית. איור 8 מציג את תוצאות הניסוי לאורך זמן. הקו האדום מייצג את הפרומפט החדש (v1.3-cot). הקו הכחול מייצג את הפרומפט הישן (v1.2). ההבדל עקבי לאורך כל הימים, מה שמחזק את המסקנה.

## 5.5 זיהוי סחיפה: Drift Detection

noitceteD tfirD
KL divergence מודד את המרחק בין שתי התפלגויות. אם $D_{KL} > 0.15$ , יש להפעיל התראה ולבדוק את הפרומפט.

כל המערכות נוטות לאי-סדר. זה לא פילוסופיה – זה החוק השני של התרמודינמיקה. גם מערכות AI סובלות מתופעה דומה. עם הזמן, הנתונים משתנים. משתמשים חדשים מתנהגים אחרת מהישנים. העולם משתנה, והמודל לא מודע לכך. זוהי "סחיפה" (drift). מחקר של Kurian ו-Allali [32] הציג שיטה לזיהוי סחיפה. הם השתמשו ב-KL divergence כמדד:

## משוואה 5.2: זיהוי סחיפה באמצעות KL Divergence



איור 8: תוצאות ניסוי A/B: השוואת גרסאות פרומפט לאורך 7 ימים

$$(15) \quad D_{KL}(P_{out}||P_{train}) = \sum_x P_{out}(x) \log \frac{P_{out}(x)}{P_{train}(x)}$$

במשוואה הזו:

-  $P_{out}$  היא התפלגות הפלטים בייצור

-  $P_{train}$  היא התפלגות הפלטים בזמן האימון

- הערך תמיד חיובי או אפס

- ערך גבוה מעיד על סחיפה

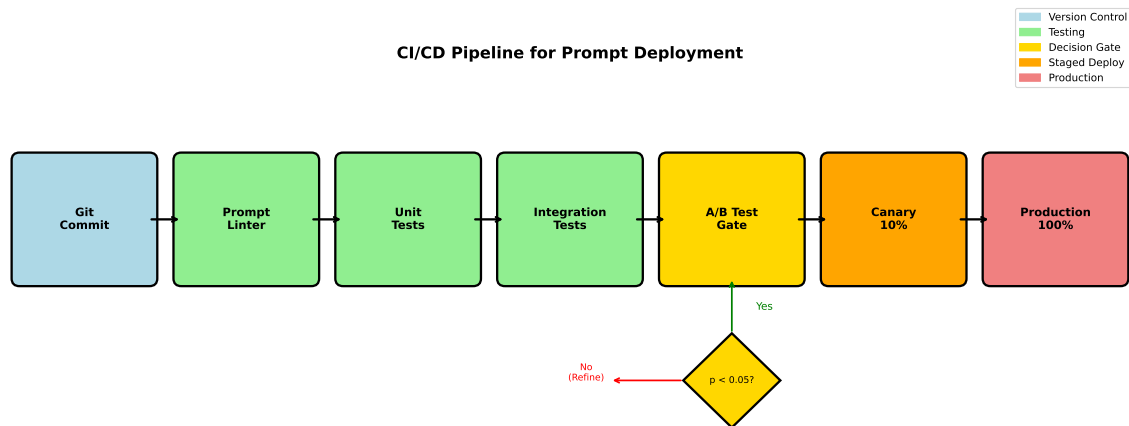
מחקר של CMU [33] בחן שיטות זיהוי בייצור. הם מצאו ש-KL divergence נותן אות מיידי כשמתרחשת סחיפה. אבל הוא רגיש לערכים קיצוניים. לכן כדאי לשלב אותו עם מדדים נוספים כמו Jensen-Shannon divergence.

## 5.6 צינור CI/CD לפרומפטים: CI/CD Pipeline

האוטומציה שחררה את בני האדם מעבודה חוזרת. היא מאפשרת להתמקד ביצירתיות. בהנדסת פרומפטים, CI/CD ממלא תפקיד דומה. איור 9 מציג את הצינור המומלץ. הוא כולל שבעה שלבים:

1. **Git Commit** – שמירת הפרומפט במערכת ניהול גרסאות

2. **Prompt Linter** – בדיקת תחביר ותבניות



איור 9: צינור CI/CD לפרומפט: משלב הפיתוח ועד לייצור

3. **Unit Tests** – בדיקות יחידה על דוגמאות

4. **Integration Tests** – בדיקות אינטגרציה עם המערכת

5. **A/B Test Gate** – שער החלטה: האם  $p\text{-value} > 0.05$ ?

6. **10% Canary** – פריסה לחלק קטן מהמשתמשים

7. **100% Production** – פריסה מלאה

מחקר של Google Cloud [34] מגדיר ארבעה עקרונות ל-MLOps: CI (Continuous Integration), CD (Continuous Delivery), CT (Continuous Training), ו-CM (Continuous Monitoring). כל העקרונות הללו רלוונטיים גם לפרומפטים.

## 5.7 אופטימיזציה עלות-דיוק: Cost-Accuracy Optimization

הכלכלה מלמדת שלכל בחירה יש מחיר. ב-AI, אנחנו סוחרים דיוק בזמן תגובה, איכות בעלות. השאלה היא: מה האיזון הנכון? טבלה 14 מציגה את המדדים החשובים. לכל מדד יש סף מומלץ ותנאי התראה. כשתנאי ההתראה מתקיים, צריך לבדוק את הפרומפט. המושג "חזית פארטו" (Pareto frontier) מתאר את האיזון האופטימלי [35]. כל נקודה על החזית היא פתרון שלא ניתן לשפר בו מדד אחד מבלי לפגוע באחר.

### משוואה 5.3: מדד יעילות Pareto

$$(16) \quad \text{Efficiency} = \frac{\text{Accuracy}}{\log(\text{Cost} + 1)} + \lambda \cdot \text{Latency}^{-1}$$

במשוואה הזו:

- Accuracy הוא הדיוק של הפרומפט

טבלה 14: מדדי הערכה לפרומפטים בייצור

ממד	נוסחה	סף	תנאי התראה
דיוק (Accuracy)	$TP/(TP+FN)$	$> 0.90$	ירידה $< 2\%$ ב-24 שעות
שיעור הזיות	$FP/(FP+TN)$	$< 0.05$	עלייה $< 1\%$
זמן תגובה P99	percentile(99)	$< 2000ms$	$< 2500$ מ"ש
עלות למשימה	tokens price	$< \$0.01$	$< 0.015\$$
ציון סחיפה	$D_{KL}$	$< 0.10$	$< 0.15$

- Cost היא העלות לבקשה

- Latency הוא זמן התגובה

-  $\lambda$  הוא משקל שמאזן את חשיבות זמן התגובה

מחקר של ParetoQ [36] הראה שאפשר לשפר את חזית פארטו. הם צמצמו את הפער בין דחיסה ל-2 ביט לבין דיוק מלא. העיקרון דומה: חפשו פתרונות שמשפרים את היעילות הכוללת.

## 5.8 דוגמאות עשה ואל תעשה: Do and Don't Examples

✓ **עשה:** Version Control

"פרומפט: v1.4-cot-manager

Tests: test\_set\_gsm8k\_hash=0x4f2a1c

Metrics: accuracy=0.912, latency\_p99=1850ms, cost=\$0.009

Deployed: 2025-12-02 06:34 UTC

"Author: yoram.segal

▲ **אל תעשה:** No Versioning

"פרומפט: גרסה אחרונה"

✓ עשה: Statistical Significance

"הגדרתי גודל מדגם  $n = 1000$ . אחרי 7 ימים,  $p\text{-value} = 0.003 < 0.05$ , אז מפרסם את הפרומפט החדש."

▲ אל תעשה: No Testing

"אני מרגיש שהפרומפט החדש טוב יותר, אז אני מפרסם אותו."

✓ עשה: Drift Monitoring

"בודק KL divergence כל 6 שעות. אתמול היה 0.08, היום  $0.15 < 0.18$ . מעדכן את הפרומפט."

▲ אל תעשה: No Monitoring

"הפרומפט עובד, לא צריך לבדוק."

✓ עשה: Cost Optimization

"גרסה v1.3-cot עולה \$0.009 לבקשה. בדקתי עם v1.3 (ללא CoT) בעלות \$0.007, דיוק רק 2% נמוך. מעבר ל-v1.3 בייצור."

▲ אל תעשה: Ignore Cost

"דיוק הכי חשוב, לא משנה העלות."

## 5.9 סיכום הפרק: Chapter Summary

בפרק זה למדנו כיצד להעביר פרומפטים מהמעבדה לייצור. הנה העקרונות המרכזיים:

1. **ניהול גרסאות:** כל פרומפט צריך version ID, מדדי ביצוע, ותאריך הטמעה.
2. **מובהקות סטטיסטית:** אל תסמכו על תחושות. השתמשו במדגם של לפחות 1000 דגימות לכל קבוצה.
3. **בדיקות A/B:** השוו את הפרומפט החדש לישן. פרסמו רק אם  $p\text{-value} > 0.05$ .
4. **זיהוי סחיפה:** עקבו אחרי KL divergence. התריעו כש- $D_{KL} > 0.15$ .
5. **צינור CI/CD:** אוטומציה של כל התהליך, משלב הפיתוח ועד לייצור.
6. **אופטימיזציית עלות-דיוק:** חפשו את נקודת האיזון על חזית פארטו.

המסר המרכזי: הנדסת פרומפטים בייצור היא מדע, לא אומנות. היא דורשת מדידה, בדיקות, ואיטרציה מתמדת. רק כך נוכל לבנות מערכות אמינות שעובדות לאורך זמן.

## 5.11 English References

- 1 P. Sahoo, A. K. Singh, S. Saha, V. Jain, S. Mondal, and A. Chadha, "A systematic survey of prompt engineering in large language models: Techniques and applications," *arXiv preprint arXiv:2402.07927*, 2024.
- 2 L. Huang et al., "A survey on hallucination in large language models: Principles, taxonomy, challenges, and open questions," *arXiv preprint arXiv:2311.05232*, 2024.
- 3 S. Tonmoy, S. Zaman, V. Jain, A. Rani, V. Rawber, and A. Chadha, "A comprehensive survey of hallucination mitigation techniques in large language models," *arXiv preprint arXiv:2401.01313*, 2024.
- 4 C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 3, 379–423, 1948.
- 5 N. F. Liu et al., "Lost in the middle: How language models use long contexts," *Transactions of the Association for Computational Linguistics*, vol. 12, 157–173, 2024. DOI: [10.1162/tac1\\_a\\_00638](https://arxiv.org/abs/2307.03172) [Online]. Available: <https://arxiv.org/abs/2307.03172>
- 6 S. Schulhoff et al., "The prompt report: A systematic survey of prompting techniques," *arXiv preprint arXiv:2406.06608*, 2024.
- 7 A. Vaswani et al., "Attention is all you need," *Advances in Neural Information Processing Systems*, vol. 30, 5998–6008, 2017. [Online]. Available: <https://arxiv.org/abs/1706.03762>
- 8 J. Wei et al., "Chain-of-thought prompting elicits reasoning in large language models," *Advances in Neural Information Processing Systems*, vol. 35, 24824–24837, 2022. [Online]. Available: <https://arxiv.org/abs/2201.11903>
- 9 T. Khot et al., "Decomposed prompting: A modular approach for solving complex tasks," in *International Conference on Learning Representations (ICLR)*, 2023. [Online]. Available: [https://openreview.net/forum?id=\\_nGgzQjzaRy](https://openreview.net/forum?id=_nGgzQjzaRy)
- 10 X. Wang et al., "Self-consistency improves chain of thought reasoning in language models," *arXiv preprint arXiv:2203.11171*, 2023.
- 11 S. Diao, P. Wang, Y. Lin, R. Pan, X. Liu, and T. Zhang, "Active prompting with chain-of-thought for large language models," in *Proceedings of the 62nd Annual Meeting of the ACL*, 2024, 1330–1350. [Online]. Available: <https://aclanthology.org/2024.acl-long.73/>

- 12 W. Chen, Y. Liu, and H. Zhang, "The few-shot dilemma: Over-prompting large language models," *arXiv preprint arXiv:2509.13196*, 2025. [Online]. Available: <https://arxiv.org/abs/2509.13196>
- 13 A. Madaan et al., "Self-refine: Iterative refinement with self-feedback," *Advances in Neural Information Processing Systems*, vol. 36, 2023. [Online]. Available: <https://arxiv.org/abs/2303.17651>
- 14 S. Yao et al., "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2023.
- 15 S. Yao et al., "Tree of thoughts: Deliberate problem solving with large language models," *arXiv preprint arXiv:2305.10601*, 2023.
- 16 A. Kong et al., "Better zero-shot reasoning with role-play prompting," *arXiv preprint arXiv:2308.07702*, 2023. [Online]. Available: <https://arxiv.org/abs/2308.07702>
- 17 M. Zheng, J. Pei, and D. Jurgens, "When a helpful assistant is not really helpful: Personas in system prompts do not improve performances of large language models," *arXiv preprint arXiv:2311.10054*, 2023. [Online]. Available: <https://arxiv.org/abs/2311.10054>
- 18 J. Kim, N. Yang, and K. Jung, "Persona is a double-edged sword: Enhancing the zero-shot reasoning by ensembling the role-playing and neutral prompts," *arXiv preprint arXiv:2408.08631*, 2024. [Online]. Available: <https://arxiv.org/abs/2408.08631>
- 19 T. Brown et al., "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, vol. 33, 1877–1901, 2020.
- 20 Anthropic. "Building effective agents. "[Online]. Available: <https://www.anthropic.com/engineering/building-effective-agents>
- 21 R. Team, "React 2.0: Enhanced reasoning and acting with reflection," *arXiv preprint*, 2025.
- 22 T. Rebedea, R. Dinu, M. Sreedhar, C. Parisien, and J. Cohen, "Nemo guardrails: A toolkit for controllable and safe llm applications with programmable rails," *arXiv preprint arXiv:2310.10501*, 2023.
- 23 M. Suzgun and A. T. Kalai, "Meta-prompting: Enhancing language models with task-agnostic scaffolding," *arXiv preprint arXiv:2401.12954*, 2024.
- 24 T. Schick et al., "Toolformer: Language models can teach themselves to use tools," in *Advances in Neural Information Processing Systems*, 36, 2023.
- 25 Y. Qin et al., "Toolllm: Facilitating large language models to master 16000+ real-world apis," *arXiv preprint arXiv:2307.16789*, 2023.

- 26 S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," *arXiv preprint arXiv:2305.15334*, 2023.
- 27 OpenAI. "Introducing structured outputs in the api. "[Online]. Available: <https://openai.com/index/introducing-structured-outputs-in-the-api/>
- 28 F. Yan et al., "Berkeley function-calling leaderboard," *arXiv preprint arXiv:2402.15491*, 2024.
- 29 L. Bernardi, T. Mavridis, and P. Estevez, "150 successful machine learning models: 6 lessons learned at booking.com," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ACM, 2019, 1743–1751. DOI: [10.1145/3292500.3330744](https://doi.org/10.1145/3292500.3330744)
- 30 R. Kumar and P. Sharma, "End-to-end mlops for scalable model deployment," *International Journal of Computer Trends and Technology*, vol. 72, no. 11, 118–125, 2024. DOI: [10.14445/22312803/IJCTT-V72I11P118](https://doi.org/10.14445/22312803/IJCTT-V72I11P118)
- 31 Apple Machine Learning Research, "All about sample-size calculations for a/b testing: Novel extensions and practical guide," *Apple Machine Learning Research*, 2024. [Online]. Available: <https://machinelearning.apple.com/research/sample-size-calculations>
- 32 J. F. Kurian and M. Allali, "Detecting drifts in data streams using kullback-leibler divergence measure for data engineering applications," *Journal of Data, Information and Management*, vol. 6, 207–216, 2024. DOI: [10.1007/s42488-024-00119-y](https://doi.org/10.1007/s42488-024-00119-y)
- 33 S. Ackermann, M. Bhardwaj, I. Grosse, and K. Hildebrand, "Augur: A step towards realistic drift detection in production ml systems," in *Workshop on Software Engineering for Responsible AI (SE4RAI)*, CMU SEI, 2022. [Online]. Available: [https://www.sei.cmu.edu/documents/614/2022\\_019\\_001\\_877199.pdf](https://www.sei.cmu.edu/documents/614/2022_019_001_877199.pdf)
- 34 Google Cloud. "Mlops: Continuous delivery and automation pipelines in machine learning. "[Online]. Available: <https://cloud.google.com/architecture/mlops-continuous-delivery-and-automation-pipelines-in-machine-learning>
- 35 H. A. R. Team, "Pareto-optimized open-source llms for healthcare via context retrieval," *arXiv preprint arXiv:2409.15127*, 2024.
- 36 Q. R. Team, "Paretoq: Improving scaling laws in extremely low-bit llm quantization," *arXiv preprint arXiv:2502.02631*, 2025.