𝟏 𝐌𝐚𝐫𝐤𝐝𝐨𝐰𝐧  
 Helps structure your prompts clearly for better AI understanding.  
 🧩 Example: Create a login function with username and password  
1.1 Markdown (Standard)  
1.2 XML  
1.3 JSON  
  
𝟐 𝐖𝐒𝐋 (Windows Subsystem for Linux)  
 A universal coding environment that allows developers to run Linux tools directly on Windows no need for dual booting! 🖥️  
  
𝟑 𝐀𝐈 𝐂𝐋𝐈  
 Command-line tools that integrate AI models into your workflow:  
 💻 Claude Code ($20/month)  
 💻 Codex CLI ($20/month)  
 💻 Gemini CLI (1000 requests free)  
 💻 QWEN CLI (2000 requests free)  
  
𝟒 𝐀𝐈 𝐂𝐋𝐈'𝐬 𝐌𝐂𝐏 𝐒𝐞𝐫𝐯𝐞𝐫  
 Connects AI CLIs with local tools making automation and model management seamless. 🤖  
  
𝟓 𝐓𝐞𝐬𝐭 𝐃𝐫𝐢𝐯𝐞𝐧 𝐃𝐞𝐯𝐞𝐥𝐨𝐩𝐦𝐞𝐧𝐭 𝐖𝐨𝐫𝐤𝐟𝐥𝐨𝐰  
 Write tests before writing code to ensure reliability and faster debugging. ✅  
  
𝟔 𝐒𝐩𝐞𝐜 𝐃𝐫𝐢𝐯𝐞𝐧 𝐃𝐞𝐯𝐞𝐥𝐨𝐩𝐦𝐞𝐧𝐭  
 Define clear business requirements before development — plan smart, build faster. 🧠  
  
𝟕 𝐃𝐞𝐩𝐥𝐨𝐲𝐦𝐞𝐧𝐭  
Bring your AI and apps to life in the cloud! 🌐  
  
𝐂𝐥𝐮𝐬𝐭𝐞𝐫: Group of computers  
🌍 Digital Ocean | ☁️ Azure  
7.1 Docker  
7.2 Kubernetes  
7.3 Dapr  
7.4 Ray (only for Linux)

What is Markdown in Prompt Engineering?

Markdown is a lightweight text formatting syntax that helps you make your prompts structured, readable, and clear for both *you* and the *AI model*.

When writing prompts for large language models (like GPT-5), Markdown is used to:

* Highlight important parts (like bold, italics, headings)
* Organize text (with bullet points, numbered lists, or tables)
* Separate sections (with dividers or code blocks)

This improves clarity, structure, and control in your prompt.

Why Markdown is Useful in Prompt Engineering

AI models (like GPT) read Markdown formatting easily.  
It helps the model differentiate between:

* Instruction
* Context
* Example Input/Output
* Desired format of response

That means Markdown = structured context engineering inside your prompt.

Prompt (with Markdown)

# Task

You are an expert data analyst.

## Input

A CSV file containing customer data with the columns: `Name`, `Age`, `City`, and `Purchases`.

## Goal

1. Summarize the data by city.

2. Find which city has the most purchases.

3. Present your result in a \*\*table\*\* format.

🤖 Model Output

| City | Total Purchases |
| --- | --- |
| Karachi | 320 |
| Lahore | 280 |
| Islamabad | 150 |

𝟐 𝐖𝐒𝐋 (Windows Subsystem for Linux)

is a compatibility layer developed by Microsoft that allows you to run a full Linux environment directly on Windows — without needing a virtual machine or dual-boot setup.

In Simple Terms

WSL lets you:

“Run Linux commands, tools, and applications *inside* Windows — as if Linux and Windows were working together.”

You can use bash, apt, python, gcc, git, node, and more — all inside Windows.

How It Works

* It provides a Linux kernel interface that runs on Windows.
* You can install Linux distributions (like Ubuntu, Debian, Kali, etc.) from the Microsoft Store.
* It runs in a lightweight virtual environment but shares the same file system and resources as Windows.

| Version | Description |
| --- | --- |
| WSL 1 | Translates Linux system calls into Windows system calls. Faster for small tasks but limited compatibility. |
| WSL 2 | Uses a real Linux kernel inside a lightweight virtual machine. Offers full system call compatibility and better performance for development. |

Common Uses

* Running Linux commands on Windows (bash, grep, ls, etc.)
* Using package managers like apt or yum
* Running Python, Node.js, or other dev environments in Linux mode
* Using Docker and containers (especially with WSL 2)
* Accessing both Windows and Linux files easily (/mnt/c/ for Windows drives)

Example: Installing Ubuntu via WSL

wsl --install -d Ubuntu

✅ You just installed and ran Python — *natively inside Windows via Linux!*

Key Benefit

You get the power of Linux (tools, scripts, servers)  
with the convenience of Windows (apps, UI, IDEs).

Concept Overview

Prompt Engineering involves designing, testing, and refining prompts to get the best output from AI models (like GPT).

When working on AI apps, chatbots, or agentic systems, developers often need Linux tools, Python environments, or open-source SDKs — all of which work more smoothly in Linux than Windows.

Goal

You want to test multiple prompts using Python scripts to evaluate LLM responses.

Without WSL

* You might face installation issues with Linux-only packages.
* Commands like pip, curl, or grep might behave differently.
* Managing virtual environments can get messy on Windows.

Why WSL Helps Prompt Engineers

| Feature | Benefit |
| --- | --- |
| 🐧 Linux environment | Works seamlessly with Python, shell scripts, and APIs |
| 📦 Package management | Use apt, pip, or conda without compatibility issues |
| 🔄 Automation | Run bash scripts to batch-test many prompts |
| 💬 Integration | Combine prompt tests with open-source tools (LangChain, LlamaIndex, etc.) |
| ⚡ Speed | Faster setup than using full virtual machines |

𝟑 𝐀𝐈 𝐂𝐋𝐈

🔹 What is AI CLI?

AI CLI stands for Artificial Intelligence Command Line Interface —  
a terminal-based tool that lets developers interact with AI models directly using commands, instead of using web dashboards or APIs manually.

Think of it as a prompting playground inside your terminal 💻 — where you can send prompts, test model behavior, and automate prompt workflows.

It’s like talking to ChatGPT, but through your command line (Terminal / PowerShell)

How It Helps in Prompt Engineering

Prompt Engineers often need to:

* Experiment with different prompt versions
* Compare model outputs quickly
* Automate prompt testing
* Integrate prompts into development pipelines

With an AI CLI, you can:

1. 🧪 Test prompts locally without opening a browser.
2. ⚙️ Automate prompt tests using scripts.
3. 🔁 Version-control prompts (using Git).
4. 📊 Benchmark outputs from different models.
5. 🚀 Deploy or share prompt templates easily.

| Concept | Description | Example |
| --- | --- | --- |
| AI CLI | Command-line tool for interacting with AI | ai chat "What is WSL?" |
| Use in Prompt Engineering | Test, refine, and automate prompts | ai prompt run prompt.md |
| Advantage | Faster experimentation, scriptable, reproducible | Automate testing prompts daily |
| Common Tools | OpenAI AI CLI, LangChain CLI, Ollama CLI | ollama run llama3 |

What is “AI CLI work” in Prompt Engineering?

AI CLI works as a bridge between you (the Prompt Engineer) and the AI model (like GPT-4o) —

allowing you to run, test, debug, and improve prompts directly from the command line.

It automates the process of sending prompts to models and getting outputs —  
so you can focus on refining the quality of prompts instead of manually copy-pasting them in a web interface.

It performs 4 main roles

1. Takes Your Prompt as Input

➡ The AI CLI reads that text prompt from your command line.  
➡ It can also load a prompt file (e.g., prompt.md or prompt.json) for complex experiments.

2. Sends It to the Model API

AI CLI connects to your chosen model endpoint (like OpenAI’s gpt-4o, Anthropic’s Claude, or a local LLM).  
It sends:

* Your prompt text
* Optional parameters (model, temperature, system message, etc.)

3. Receives and Displays the Model Output

The CLI then prints the model’s reply instantly in your terminal.  
This makes it easy to:

* Observe different outputs
* Compare prompt changes
* Log results into files for analysis

4. Supports Iteration and Automation

In Prompt Engineering, testing multiple versions of a prompt is key.  
AI CLI lets you automate and compare versions quickly:

Why AI CLI Is Powerful for Prompt Engineers

| Feature | Benefit |
| --- | --- |
| 🔤 Direct Prompt Input | Instantly test new ideas |
| 🧩 File + Template Support | Manage structured prompts (system + user) |
| 📊 Automation | Benchmark or A/B-test prompts |
| 🔒 Reproducibility | Same setup, same output |
| 🧰 Integration | Connects with version control, CI/CD, and notebooks |

AI CLI works as a fast testing engine for prompts.  
It automates the *prompt → model → response → evaluation* loop  
— the core cycle of Prompt Engineering.

𝐀𝐈 𝐂𝐋𝐈'𝐬 𝐌𝐂𝐏 𝐒𝐞𝐫𝐯𝐞𝐫

🧠 What is **AI CLI’s MCP Server?**

**AI CLI (Command Line Interface)** allows developers to **interact with AI models directly from the terminal**, using commands like ai run, ai ask, etc.

Inside this system, **MCP** stands for: Model Context Protocol

So the **AI CLI’s MCP Server** is a **background process** or **server** that follows the **Model Context Protocol**, enabling the CLI (and other tools) to:

* Communicate with AI models,
* Manage context (memory, data, tools),
* Handle requests (like retrieval, summarization, generation, etc.),
* Maintain consistent behavior across different tools or plugins.

The **MCP Server** acts as a **bridge** between:

* 🧑‍💻 **You (the user)** typing commands in the CLI,
* 🤖 **The AI model**, which processes and generates results.

It ensures that:

1. The model receives the right **context** (like files, documents, or chat history),
2. The request and response follow a **standard protocol** (structured JSON or schema),
3. The interaction is **reproducible** and **consistent** across environments.

Using MCP Server via AI CLI

Suppose you have an **AI CLI** tool installed and configured with an **MCP Server** to summarize documents.

Command in CLI:

ai run summarize --file report.pdf

What happens internally:

The **AI CLI** sends a **“summarize” request** to the **MCP Server**.

The **MCP Server** attaches the file context (report.pdf) and prepares a structured request.

The server uses the connected **AI model** to process the task.

The MCP Request (Standard Format Example):

{

"action": "summarize",

"input": {

"file": "report.pdf",

"context": "business\_report"

},

"model": "gpt-4o",

"metadata": {

"temperature": 0.3,

"max\_tokens": 500

}

}

4️⃣ The MCP Response:

{

"status": "success",

"output": {

"summary": "The report covers Q2 sales performance, highlighting a 15% growth in revenue..."

},

"usage": {

"tokens\_used": 250

}

}

𝐓𝐞𝐬𝐭 𝐃𝐫𝐢𝐯𝐞𝐧 𝐃𝐞𝐯𝐞𝐥𝐨𝐩𝐦𝐞𝐧𝐭 𝐖𝐨𝐫𝐤𝐟𝐥𝐨𝐰

**What is Test-Driven Development (TDD)?**

**Test-Driven Development (TDD)** is a **software engineering approach** where you:

1. **Write a test** first (before writing the code),
2. **Run the test** to see it fail,
3. **Write the minimal code** to make the test pass,
4. **Refactor** and improve the code while keeping the test passing.

**Applying TDD to Prompt Engineering**

In **Prompt Engineering**, instead of writing **code functions**, you’re writing **prompts** that instruct AI models to produce correct outputs.

So **TDD in Prompt Engineering** means:

You write *test cases* to verify your prompt gives the **desired and consistent response**, then refine the prompt until the “tests” pass.

**TDD Workflow in Prompt Engineering**

Here’s the **step-by-step standard workflow**:

| **Step** | **Description** | **Example** |
| --- | --- | --- |
| **1. Write a test (expected output)** | Define what kind of output you want from the prompt. | “AI should return a 2-paragraph summary of a text.” |
| **2. Write the prompt** | Create a prompt that should ideally produce that output. | “Summarize the following article in exactly 2 paragraphs with clear key points.” |
| **3. Run the prompt** | Send it to the model and see the result. | The AI returns a 3-paragraph summary. ❌ |
| **4. Check test results** | Compare actual vs. expected output. | Fail: The output is too long and not structured. |
| **5. Refine the prompt** | Modify the wording or add constraints. | “Summarize the article in *two concise paragraphs*, focusing only on main ideas.” |
| **6. Re-run the test** | Test again until AI passes your criteria. | ✅ Now AI gives 2 clear paragraphs. |
| **7. Automate (optional)** | Use tools to automate these tests (e.g., eval frameworks like promptfoo, LangSmith, or OpenAI Evals). | Automated tests check model consistency across samples. |

𝐒𝐩𝐞𝐜 𝐃𝐫𝐢𝐯𝐞𝐧 𝐃𝐞𝐯𝐞𝐥𝐨𝐩𝐦𝐞𝐧𝐭

What is **Spec-Driven Development (SDD)?**

**Spec-Driven Development** means you start by writing a **specification (spec)** — a *clear, detailed description* of **what the system or model must do**, **how it must respond**, and **what standards the output must meet**.

simple words:  
You write a “contract” first, then design prompts (or code) that strictly follow that contract.

In **Prompt Engineering**, **Spec-Driven Development (SDD)** means:

You define a **specification** for the AI’s output *before* writing or testing your prompt.  
Then you create prompts that **comply with this spec** and can be **verified automatically or manually**.

Standard Spec-Driven Workflow in Prompt Engineering

| **Step** | **Description** | **Example** |
| --- | --- | --- |
| **1. Define Specification (Spec)** | Describe exactly how the model should respond: structure, tone, format, and constraints. | “The output must include a title, summary (≤100 words), and 3 bullet points.” |
| **2. Write Prompt Based on Spec** | Create a prompt that instructs the model to follow the spec. | “Summarize the article. Output format: Title, Summary (under 100 words), and 3 Key Points.” |
| **3. Run Prompt** | Send it to the AI model. | The AI returns an output. |
| **4. Validate Against Spec** | Check if the output matches the specification (format, tone, and length). | If any condition fails → revise prompt. |
| **5. Refine Prompt or Spec** | Adjust until the model always meets the spec. | Add stricter wording, examples, or schema. |
| **6. Automate (Optional)** | Use tools or evals to automatically verify compliance with the spec. | Prompt evals check if output meets schema. |

Create a prompt that makes AI generate **structured interview summaries** for HR teams.

You define exactly how the AI should respond.

Step 1️ Define Specification (The “Spec”)

spec:

title: "Interview Summary Spec"

output\_structure:

- Candidate Name: string

- Role Applied: string

- Strengths: list of 3 bullet points

- Weaknesses: list of 2 bullet points

- Final Verdict: one sentence summary

constraints:

- Tone: professional

- Length: under 150 words

Step 2️ Write the Prompt

You are an HR assistant. Generate a structured summary of the interview in the following format:

Candidate Name: <name>

Role Applied: <role>

Strengths:

- <point 1>

- <point 2>

- <point 3>

Weaknesses:

- <point 1>

- <point 2>

Final Verdict: <1-sentence professional conclusion>

Step 3️ Run the Prompt

“Interview with Ayesha Khan for the Software Engineer role. She showed great problem-solving skills and teamwork, but lacks leadership experience.”

Step 4️ Validate Against Spec

Structure? ✅

Bullet counts correct? ✅

Tone professional? ✅

Word count < 150? ✅  
✅ Test Passed.

**Analogy**

🔧 **TDD** = "Check if the car drives properly on the road."  
🏗️ **SDD** = "Design the blueprint of the car before building it."

**TDD** helps you verify correctness of AI responses.

**SDD** ensures all AI outputs have consistent **format**, **fields**, and **style** — like API or data schema compliance.

**Deployment**

What is **Deployment in Prompt Engineering?**

In **Prompt Engineering**,

**Deployment** means taking a well-tested and refined **prompt (or prompt system)** and making it **available for real users, applications, or production use**.

In other words:  
After you design, test, and evaluate your prompts, **deployment** is the final step — where your prompt runs automatically inside an app, chatbot, or API workflow.

| **Software Development** | **Prompt Engineering** |
| --- | --- |
| Deploying code to a web server | Deploying prompts into an AI model workflow |
| Your code runs for users | Your prompt interacts with real users via AI |
| You monitor performance & errors | You monitor response quality & model behavior |

**Standard Prompt Engineering Deployment Workflow**

| **Step** | **Stage** | **Description** |
| --- | --- | --- |
| **1️ Design** | **Create and structure your prompt** | **Define purpose, input, output format, and instructions** |
| **2️ Test & Evaluate** | **Check for reliability** | **Use multiple test inputs and compare outputs** |
| **3️ Refine** | **Improve based on results** | **Adjust wording, temperature, examples, or context** |
| **4️ Version & Approve** | **Freeze a stable version** | **Save as “Prompt v1.0” or similar** |
| **5️ Deploy** | **Integrate into an app or API** | **Plug into your chatbot, workflow, or backend** |
| **6️ Monitor** | **Track real-time performance** | **Collect metrics: accuracy, user satisfaction** |
| **7️ Iterate** | **Update periodically** | **Improve prompt versions based on feedback** |

Deploying a Prompt in Real Workflow

Deploy a **Customer Support Assistant** prompt that answers user queries politely and concisely.

Step 1️– Design Prompt

You are a polite and professional customer support assistant.

Answer the user’s question clearly in under 80 words.

If the question is unclear, ask for more details.

Step 2️ – Test in Sandbox

You test this prompt manually or in an evaluation tool:

* Input: “My order hasn’t arrived yet.”
* Output: “I’m sorry to hear that. Could you please share your order ID so I can check its status?”  
  ✅ Works well.

Step 3️ – Finalize Version

You save the prompt as a version:

prompt\_support\_v1

Step 4️ – **Deploy via API**

You integrate it into your app using an AI SDK (e.g., OpenAI or Anthropic):

from openai import OpenAI

client = OpenAI()

def get\_support\_response(user\_message):

prompt = """

You are a polite and professional customer support assistant.

Answer the user’s question clearly in under 80 words.

If the question is unclear, ask for more details.

"""

response = client.chat.completions.create(

model="gpt-4o",

messages=[

{"role": "system", "content": prompt},

{"role": "user", "content": user\_message}

]

)

return response.choices[0].message.content

Step 5️⃣ – Monitor and Improve

You collect feedback and logs:

* If users complain answers are too short, adjust prompt length.
* If tone feels robotic, add “Use friendly and natural tone.”

You then deploy prompt\_support\_v2.

**Analogy**

✍️ **Prompt Design** = Writing your speech  
🧪 **Testing** = Practicing your speech  
🚀 **Deployment** = Giving your speech to a live audience

𝐂𝐥𝐮𝐬𝐭𝐞𝐫

A **cluster** is a **group of interconnected computers (or systems)** that work together as if they were **a single powerful system**.

Each computer in the cluster is called a **node**, and all nodes cooperate to perform tasks **faster, more reliably, and at larger scale** than a single computer.

In **Prompt Engineering**, a **Cluster** means a **group of related prompts** that focus on the **same main goal or topic**, but each prompt has **a slightly different style, tone, or perspective**.

Think of a **Cluster** as a *team of prompts* — all working toward one goal but in different ways.  
It helps you explore, test, and refine prompts to get the **best response** from the AI.

Imagine you have one small computer that takes 1 hour to analyze data.  
If you connect **10 computers in a cluster**, they can share the workload and finish the same job in **a few minutes**.

So a **cluster** = teamwork of computers 💻💻💻 working as one brain 🧠.

Types of Clusters

| **Type** | **Purpose** | **Example** |
| --- | --- | --- |
| **Compute Cluster** | For running heavy calculations | Scientific simulations, AI training |
| **Storage Cluster** | For storing and managing large data | Google Drive, AWS S3 |
| **Load-Balancing Cluster** | For distributing user traffic | Web servers |
| **High-Availability Cluster** | For preventing downtime | Banking or hospital systems |
| **Database Cluster** | For handling big data queries | MySQL cluster, MongoDB replica set |

In AI & Prompt Engineering Context

In **AI systems**, a **cluster** is used to:

* Run **large language models (LLMs)** across multiple GPUs or servers.
* Manage **distributed inference** (AI responses across nodes).
* Store and process large **context or retrieval data** for RAG (Retrieval-Augmented Generation).

**Example:**

When you ask ChatGPT or Gemini a question:

* Your query doesn’t go to one computer.
* It goes to a **cluster of AI servers**, each handling a small part of processing, caching, and data retrieval.
* The combined cluster gives you a fast and accurate answer.

**cluster** is like a **team of chefs** in one kitchen —  
each works on different dishes 🍲🍛,  
but together they serve the full meal quickly 🍽️.

## What is **DigitalOcean**?

**DigitalOcean** is a **cloud infrastructure company** mainly designed for **developers, startups, and small to medium businesses**.

It provides **simple, fast, and affordable cloud computing** — you can create virtual servers (called **Droplets**) in seconds.

### Main Services

* Virtual servers (Droplets) 💻
* Managed Databases (PostgreSQL, MySQL, MongoDB)
* Object Storage (Spaces)
* Kubernetes Clusters 🧩
* App Platform (for web deployment)
* Firewalls, Monitoring, and Load Balancers

What is **Microsoft Azure**?

**Azure** is **Microsoft’s cloud platform** — a **large-scale enterprise-level** service used by **corporations, governments, and developers** worldwide.

It’s a full **cloud ecosystem** that supports AI, machine learning, data analytics, IoT, virtual machines, and more.

### Main Services

* Virtual Machines (VMs)
* Azure AI / OpenAI Services 🤖
* Databases (SQL, Cosmos DB)
* Kubernetes Services (AKS)
* DevOps tools
* Big Data & Analytics
* Networking, Security, and Identity tools

## What is Docker?

**Docker** is a tool that lets you **package and run software in containers** — lightweight, isolated environments that work the same on any computer or cloud.

### Main Services

1. **Docker Engine**

2. **Docker Images**

3. **Docker Containers**

4. **Docker Hub**

5. **Docker Compose**

6. **Docker Swarm (Clustering Service)**

7. **Docker Desktop**

8. **Docker Registry**

## What is **Kubernetes**?

**Kubernetes (K8s)** is an **open-source system** for **automating the deployment, scaling, and management** of **containerized applications** (like Docker containers).

Kubernetes Main Services / Components

| **Service / Component** | **Purpose / Description** |
| --- | --- |
| 🧩 **Pods** | The smallest deployable unit in Kubernetes. Each pod can contain one or more containers (like Docker containers) that run together. |
| 💻 **Nodes** | The worker machines (physical or virtual) where Kubernetes runs your Pods. |
| 🌐 **Cluster** | A group of Nodes controlled by Kubernetes — your entire running environment. |
| 🗂️ **Namespaces** | Logical divisions inside a cluster to separate environments or teams (like folders). |
| 🚀 **Deployments** | Used to manage Pods — handles scaling (replicas), updates, and rollbacks automatically. |
| 🔁 **ReplicaSets** | Ensures the desired number of identical Pods are always running (auto-healing). |
| 🔗 **Services** | Provides stable IP addresses and DNS names to access Pods; also load balances traffic. |
| 🌍 **Ingress** | Controls external (HTTP/HTTPS) access to the services — acts like a web gateway. |
| ⚙️ **ConfigMaps** | Stores non-confidential configuration data (like environment variables). |
| 🔐 **Secrets** | Stores sensitive data securely (like API keys, passwords). |
| 💾 **Volumes / Persistent Volumes (PV)** | Provides persistent storage for Pods so data isn’t lost if a container restarts. |
| 📅 **Scheduler** | Decides on which Node each Pod should run based on resources and rules. |
| 👁️ **Controller Manager** | Monitors the cluster and keeps the actual state matching the desired state. |
| 💬 **API Server** | The central “brain” — all commands (kubectl or dashboard) communicate through it. |
| 🧠 **etcd** | A key-value database that stores the entire cluster’s configuration and state. |
| 🧩 **Kubelet** | Runs on each Node; ensures containers are running as expected. |
| 🌐 **Kube Proxy** | Handles network traffic routing and load balancing inside the cluster. |

## What is **Dapr**?

**Dapr** stands for **Distributed Application Runtime**.

It is an **open-source runtime** that helps developers build **microservices** and **cloud applications** easily and reliably — without needing to manage complex infrastructure.

**Dapr** provides a set of **building blocks (APIs and services)** that make it easier to connect, communicate, and scale distributed apps.

You can use Dapr with any programming language (Python, Node.js, Java, C#, etc.) and it runs anywhere — **locally**, **on Docker**, **Kubernetes**, or **clouds** like Azure and DigitalOcean.

## Why Dapr Exists

In large systems (microservices or AI-based apps), developers often face:

* Service-to-service communication problems
* State management (saving data)
* Messaging between components
* Security and scaling challenges

Instead of writing all that code manually, **Dapr handles it automatically** using standard APIs.

Dapr Main Services (Building Blocks)

| **Service / Building Block** | **Purpose / Description** |
| --- | --- |
| 🧩 **Service Invocation** | Allows one microservice to easily call another (with retry, encryption, tracing). |
| 📬 **Pub/Sub Messaging** | Enables event-driven communication between services (publish and subscribe model). |
| 💾 **State Management** | Simplifies storing and retrieving state (e.g., user sessions, AI data) in databases. |
| 📦 **Bindings** | Connects your app to external systems (e.g., message queues, file storage, APIs). |
| 🔐 **Secrets Management** | Securely stores and retrieves secrets (passwords, tokens, API keys). |
| 📈 **Observability** | Provides tracing, logging, and metrics to monitor app health. |
| 🗺️ **Actors (Virtual Actors)** | Simplifies building stateful, scalable objects — great for chatbots or IoT devices. |
| 🧭 **Configuration API** | Central place to manage dynamic configurations for all microservices. |
| 🧱 **Workflow API (new)** | Helps define and execute workflows across multiple services. |