

## Part A — Theory (Short Questions) Answer in your own wording

### 1. Nine Pillars Understanding

- Why is using AI Development Agents (like Gemini CLI) for repetitive setup tasks better for your growth as a system architect?

Using AI Development Agents like Gemini CLI for repetitive setup tasks is better for your growth as a system architect because it **frees up your time and mental energy**. Instead of spending hours on boilerplate code, environment configuration, or dependency setup, you can focus on **high-level system design, architecture decisions, and solving complex problems**.

This shift allows you to:

- **Think strategically** about how systems interact and scale.
- **Experiment faster**, since setup is automated and consistent.
- **Learn more deeply** by focusing on architecture patterns, trade-offs, and performance.
- **Collaborate better**, as AI agents standardize workflows and reduce human error.

In short, AI agents handle the groundwork so you can grow into a more thoughtful, creative, and impactful architect.

- Explain how the Nine Pillars of AIDD help a developer grow into an M-Shaped Developer

The Nine Pillars of AI-Driven Development (AIDD) act like a growth framework that helps developers evolve into **M-Shaped Developers** professionals who are not only deep in technical skills but also broad in strategic thinking, creativity, and collaboration.

Here's how each pillar contributes to that transformation:

### 1. Prompt Engineering

- Teaches developers how to communicate effectively with AI.
- Builds skills in abstraction, clarity, and intent—key traits of strategic thinkers.

### 2. Evaluation-Driven Development (EvDD)

- Encourages critical thinking and judgment.
- Developers learn to assess AI outputs, refine them, and make informed decisions.

### 3. Tooling & Automation

- Reduces time spent on repetitive tasks.
- Frees up mental space for innovation and architectural thinking.

### 4. System Design

- Strengthens architectural skills by focusing on how components interact.
- Promotes scalable, modular thinking—core to M-shaped depth.

## 5. Data & Feedback Loops

- Builds awareness of how systems learn and evolve.
- Encourages developers to think in terms of continuous improvement and user impact.

## 6. Human-AI Collaboration

- Enhances communication and teamwork across disciplines.
- Developers learn to guide AI and work alongside it, not just use it.

## 7. Creativity & Experimentation

- Sparks innovation by lowering the cost of trying new ideas.
- Encourages developers to explore beyond traditional boundaries.

## 8. Metrics & Impact

- Focuses on outcomes, not just output.
- Developers learn to measure success in terms of user value and system performance.

## 9. Ethics & Responsibility

- Instills a sense of accountability in how AI is used.
- Encourages thoughtful, inclusive, and safe development practices.

Together, these pillars help developers grow **vertically** in technical mastery and **horizontally** in strategic, ethical, and creative dimensions—exactly what defines an M-Shaped Developer.

## 2. Vibe Coding vs Specification-Driven Development

- Why does Vibe Coding usually create problems after one week?

**Vibe Coding**—where developers code based on intuition or "what feels right" in the moment—often leads to problems after a week because it **lacks structure, clarity, and long-term thinking**.

Here's why it breaks down:

- **No clear documentation:** Without specs, it's hard for others (or even your future self) to understand the purpose or logic behind the code.
- **Inconsistent logic:** Decisions made on the fly can lead to messy, unscalable code that's hard to maintain or extend.
- **Poor collaboration:** Team members struggle to align when there's no shared understanding or blueprint to follow.
- **Debugging becomes a nightmare:** Without a clear plan, tracking down bugs or unintended behavior becomes time-consuming and frustrating.

- **How would Specification-Driven Development prevent those problems?**

In contrast, **Specification-Driven Development** sets a clear direction from the start, making it easier to build reliable, maintainable, and collaborative systems. Vibe Coding might feel fast at first but it often leads to **technical debt** that slows everything down later.

### 3. Architecture Thinking

- **How does architecture-first thinking change the role of a developer in AIDD?**

**Architecture-first thinking transforms the role of a developer in the AI-Driven Development (AIDD) era from a code producer to a strategic system designer.**

Here's how the shift plays out:

#### **From Building Blocks to Blueprinting**

Instead of jumping straight into writing code, developers start by designing the **structure, flow, and interactions** of the system. This means thinking about scalability, modularity, and how AI components fit into the bigger picture.

#### **Higher-Level Decision Making**

Developers become more focused on **why** things are built a certain way—not just how. They evaluate trade-offs, choose the right AI tools, and define feedback loops that guide intelligent behavior.

#### **AI as a Design Partner**

With architecture-first thinking, developers learn to **orchestrate AI agents** like Gemini CLI or Copilot—not just use them. They design systems where AI handles setup, testing, and even optimization, while humans guide intent and strategy.

#### **Continuous Evolution**

Architecture-first developers build systems that are **adaptive and resilient**. They think in terms of feedback loops, data flows, and evaluation metrics—ensuring the system learns and improves over time.

#### **Collaboration and Clarity**

This mindset also improves team collaboration. Clear architecture means everyone—from designers to AI agents—can align on goals, roles, and responsibilities.

In short, architecture-first thinking elevates developers into **M-shaped leaders** who combine deep technical skills with broad strategic vision. It's the foundation for building intelligent, scalable, and future-ready systems in the AIDD era.

- Explain why developers must think in layers and systems instead of raw code.

Developers must think in **layers and systems** instead of just raw code because modern software is no longer just about writing functions—it's about building **complex, interconnected ecosystems**.

## 1. Scalability and Maintainability

- Raw code might work for small tasks, but layered thinking ensures your system can **scale** and be **maintained** over time.
- Layers like data, logic, and presentation help isolate concerns and make updates easier.

## 2. Modularity and Reusability

- Systems thinking encourages **modular design**, where components can be reused, replaced, or upgraded without breaking everything.
- It's like building with LEGO blocks instead of pouring concrete.

## 3. Better Problem Solving

- Thinking in systems helps you understand **how parts interact**, where bottlenecks might occur, and how to optimize performance.
- You're not just solving a coding problem—you're solving a **design challenge**.

## 4. AI Collaboration

- AI tools thrive when given **structured, layered input**. They can automate setup, testing, and optimization more effectively when the system is well-organized.
- Developers become orchestrators of intelligent workflows, not just coders.

## 5. Team Alignment

- Layered systems make it easier for teams to collaborate. Designers, backend engineers, and AI agents can work in parallel when the architecture is clear.
- Everyone knows where their piece fits.

## Part B — Practical Task (Screenshot Required)

**Task:** Using any AI CLI tool, generate a 1-paragraph specification for an email validation function. Requirements: Must contain “@” Must contain a valid domain (e.g., .com, .org) Should return clear error messages

### Your exact CLI prompt (text or screenshot)

Gemini Create a function called **email\_validator.py** that checks whether a given input string is a valid email address. The function must meet the following requirements: The email must contain exactly one “@” symbol, separating the local part from the domain. The domain must end with a valid top-level domain (TLD), such as .com, .org, .net or .edu. If the input fails any of these checks, the function should return a clear and specific error message, such as: “Missing '@' symbol”, “Email must contain only one '@' symbol”, “Invalid domain format”, If the email passes all checks, return a success message like “Valid email address.” The function should be modular and easy to test, with clean and readable code.

### The 1-paragraph specification generated by the CLI



```

> GEMINI
Gemini Create a function called email_validator.py that checks whether a given input string is a valid email address. The function must meet the following requirements: The email must contain exactly one "@" symbol, separating the local part from the domain. The domain must end with a valid top-level domain (TLD), such as .com, .org, .net or .edu. If the input fails any of these checks, the function should return a clear and specific error message, such as: "Missing '@' symbol", "Email must contain only one '@' symbol", "Invalid domain format", If the email passes all checks, return a success message like "Valid email address." The function should be modular and easy to test, with clean and readable code.

+ I will create a Python function to validate email addresses based on your requirements. I'll put this function in a new file named email_validator.py.

Here's my plan:
1. Create 'email_validator.py': This file will contain the validate_email function.
2. Implement 'validate_email' function:
   * Check for the count of '@' symbols.
   * Split the email into local and domain parts.

* Split the email into local and domain parts.
* Validate the domain for a '.' and an allowed TLD (.com, .org, .net, .edu).
* Return appropriate success or error messages.

I will now create the email_validator.py file.

✓ WriteFile Writing to email_validator.py

1 def validate_email(email_string: str) -> str:
2     """
3         Checks if a given input string is a valid email address based on specific rules.
4
5     Args:
6         email_string: The email address string to validate.
7
8     Returns:
9         A success message "Valid email address." if all checks pass,
10        otherwise a specific error message.
11    """
12    if email_string.count('@') == 0:
13        return "Missing '@' symbol"
14    elif email_string.count('@') > 1:
15        return "Email must contain only one '@' symbol"
16    local_part, domain_part = email_string.split('@')
17    if '.' not in domain_part:
18        return "Invalid domain format"

You are running Gemini CLI in your home directory. It is recommended to run in a project-specific directory.

```

```
✓ WriteFile Writing to email_validator.py

18     if '..' not in domain_part:
19         return "Invalid domain format"
20
21     # Split domain to check TLD
22     domain_parts = domain_part.split('.')
23     if len(domain_parts) < 2: # Ensure there's at least a domain name and a TLD
24         return "Invalid domain format"
25
26     tld = domain_parts[-1]
27     allowed_tlds = ["com", "org", "net", "edu"]
28
29     if tld not in allowed_tlds:
30         return "Invalid domain format"
31
32     return "Valid email address."
33
34 if __name__ == "__main__":
35     # Test cases
36     print(f"test@example.com: {validate_email('test@example.com')}")
37     print(f"user.name@sub.example.org: {validate_email('user.name@sub.example.org')}")
38     print(f"invalid-email: {validate_email('invalid-email')}")
39     print(f"multiple@symbols.com: {validate_email('multiple@symbols.com')}")
40     print(f"user@nodotcom: {validate_email('user@nodotcom')}")
41     print(f"user@example.xyz: {validate_email('user@example.xyz')}")
42     print(f"user@example.co.uk: {validate_email('user@example.co.uk')}")
43     print(f"user@example.edu: {validate_email('user@example.edu')}")

You are running Gemini CLI in your home directory. It is recommended to run in a project-specific directory.
```

```
✓ WriteFile Writing to email_validator.py

40     print(f"multiple@symbols.com: {validate_email('multiple@symbols.com')}")
41     print(f"user@nodotcom: {validate_email('user@nodotcom')}")
42     print(f"user@example.xyz: {validate_email('user@example.xyz')}")
43     print(f"user@.com: {validate_email('user@.com')}")
44     print(f"@example.com: {validate_email('@example.com')}")
45     print(f"example.com: {validate_email('example.com')}")
46     print(f"user@example.co.uk: {validate_email('user@example.co.uk')}")
47     print(f"user@example.edu: {validate_email('user@example.edu')}")

email_validator.py is created with the function and tests. I'll run it to confirm.
```

```
✓ Shell py email_validator.py [current working directory C:\Users\Mustafa Qazi] (Run the Python script using the 'py' launcher, common on Windows.)
'test@example.com': Valid email address.
'user.name@sub.example.org': Valid email address.
'invalid-email': Missing '@' symbol
'multiple@symbols.com': Email must contain only one '@' symbol
'user@nodotcom': Invalid domain format
'user@example.xyz': Invalid domain format
'user@.com': Valid email address.
'@example.com': Valid email address.
'example.com': Missing '@' symbol
'user@example.co.uk': Invalid domain format
'user@example.edu': Valid email address.
```

♦ The py command worked, but the email validation has issues: it incorrectly validates empty local parts and invalid domain formats like user@.com and @example.com. I need to modify validate\_email to check for an empty local\_part and ensure the domain\_part has at least two segments. I'll update email\_validator.py with clearer error messages: "Email must have a local part before '@'" and "Invalid domain format" for these cases.

```
✓ Edit email_validator.py:    if email_string.count('@')... =>    if email_string.count('@')...
16
17     local_part, domain_part = email_string.split('@')
18
```

♦ The py command worked, but the email validation has issues: it incorrectly validates empty local parts and invalid domain formats like user@.com and @example.com. I need to modify validate\_email to check for an empty local\_part and ensure the domain\_part has at least two segments. I'll update email\_validator.py with clearer error messages: "Email must have a local part before '@'" and "Invalid domain format" for these cases.

```
✓ Edit email_validator.py:    if email_string.count('@')... =>    if email_string.count('@')...
16
17     local_part, domain_part = email_string.split('@')
18
19 + if not local_part:
20 +     return "Email must have a local part before '@'"
21 +
22     if '..' not in domain_part:
23         return "Invalid domain format"
24
25     # Split domain to check TLD
26     domain_parts = domain_part.split('.')
27     if len(domain_parts) < 2: # Ensure there's at least a domain name and a TLD
28 +     if len(domain_parts) < 2 or any(not part for part in domain_parts): # Ensure there's at least a domain name and a TLD, and no empty parts
29         return "Invalid domain format"
30
31     tld = domain_parts[-1]
```

+ Modified email\_validator.py. Rerunning script to verify.

```
+ Modified email_validator.py. Rerunning script to verify.  
✓ Shell py email_validator.py [current working directory C:\Users\Mustafa Qazi] (Run the Python script to test the updated email validation function.)  
'test@example.com': Valid email address.  
'user.name@sub.example.org': Valid email address.  
'invalid-email': Missing '@' symbol  
'multiple@symbols.com': Email must contain only one '@' symbol  
'user@nodotcom': Invalid domain format  
'user@example.xyz': Invalid domain format  
'user@.com': Invalid domain format  
'@example.com': Email must have a local part before '@'  
'example.com': Missing '@' symbol  
'user@example.co.uk': Invalid domain format  
'user@example.edu': Valid email address.  
+ The updated script's output is satisfactory. Both 'user@.com' and '@example.com' now correctly return "Invalid domain format" and "Email must have a local part before '@'" respectively. All test cases align with requirements, confirming the validate_email function is robust. Task complete.  
✓ Write Todos Set 1 todo(s)  
+ I have completed the task. The email_validator.py file contains the validate_email function, which now correctly validates email addresses according to all specified requirements and provides clear error messages. I have also verified its functionality with test cases.
```

## **Part C — Multiple Choice Questions**

### **1. What is the main purpose of Spec-Driven Development?**

- A. Make coding faster
- B. Clear requirements before coding begins**
- C. Remove developers
- D. Avoid documentation

### **2. What is the biggest mindset shift in AI-Driven Development?**

- A. Writing more code manually
- B. Thinking in systems and clear instructions**
- C. Memorizing more syntax
- D. Working without any tools

### **3. Biggest failure of Vibe Coding?**

- A. AI stops responding
- B. Architecture becomes hard to extend**
- C. Code runs slow
- D. Fewer comments written

### **4. Main advantage of using AI CLI agents (like Gemini CLI)?**

- A. They replace the developer completely
- B. Handle repetitive tasks so dev focuses on design & problem-solving**
- C. Make coding faster but less reliable
- D. Make coding optional

**5. What defines an M-Shaped Developer?**

- A. Knows little about everything
- B. Deep in only one field
- C. Deep skills in multiple related domains**
- D. Works without AI tools

**Reflection**

This task helped me see how quickly the role of a developer is changing in today's AI-native world. Ideas like the Nine Pillars, Spec-Driven Development, and AI agents aren't just helpful tools—they're part of a bigger transformation in how we design, build, and manage software. By focusing on architecture from the start and learning to work alongside AI tools, developers can move beyond just writing code. We start thinking in systems, solving bigger problems, and developing deep expertise across multiple areas. That's what it means to grow into an M-Shaped Developer—someone who blends strong technical skills with smart, AI-powered workflows.