

## Hands-On Lab

# Task Decomposition, Planning, Implementation, Review & Verification Patterns

This lab is designed for software developers, AI engineers, and product managers who want to transition from unpredictable code generation to a structured, highly reliable agentic workflow. By completing this exercise, you will master five core engineering patterns: Decomposition, Planning, Implementation, Review, and Verification- by guiding Claude Code to build and test a production-ready email validator, establishing an end-to-end template you can confidently reuse for all future AI-assisted feature development.

### **Real-Time Scenario:**

Software Developers, AI/ML Engineers, and Product Managers need to move beyond unpredictable "vibe-coding" and adopt a structured approach to agentic AI development. When AI generates code without a plan, it often misses edge cases, hallucinates APIs, or produces untested logic, leading to costly debugging. In this lab scenario, learners build a small feature (an email validator) not by asking the AI to just "write the code," but by guiding it through five professional engineering patterns: Decomposition, Planning, Implementation, Review, and Verification. This structured workflow ensures reliable, bug-free, and production-ready outputs.

### **Description:**

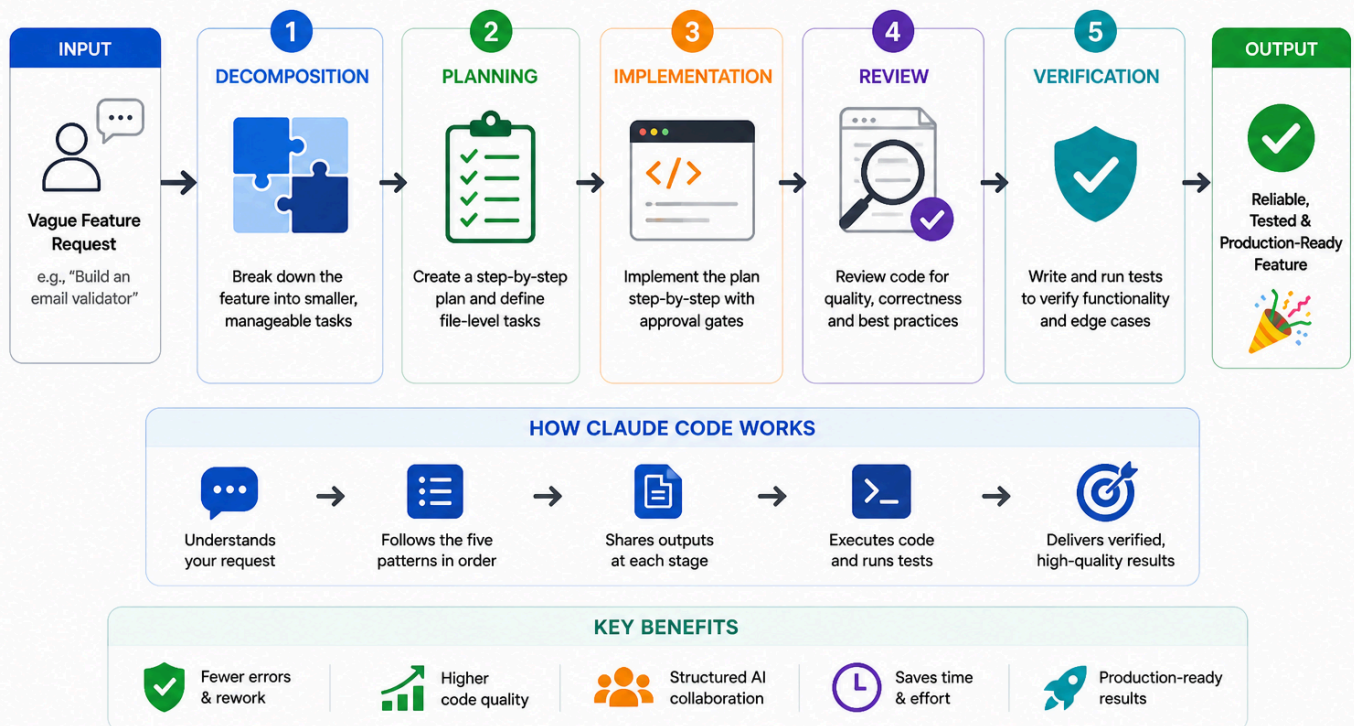
This lab focuses on the five core patterns every agentic developer must master to turn Claude Code from a basic code generator into a reliable engineering collaborator. Learners will start by setting up a fresh Node.js project. They will then use specific prompt patterns to force Claude Code to break down a vague feature request into sub-tasks, generate an explicit file-level plan, implement the code step-by-step with strict approval gates, perform self and peer reviews, and finally write and execute a Jest test suite. By mastering these patterns, learners establish an end-to-end workflow template they can reuse for every future AI-assisted feature.

## Contents

<b>1 Introduction.....</b>	<b>3</b>
<b>2 Overview &amp; Key Concepts.....</b>	<b>4</b>
2.1 Why Patterns Matter in Agentic Development.....	4
2.2 The Five Core Patterns.....	4
2.3 How Claude Code Executes These Patterns.....	5
<b>3 Documentation Links.....</b>	<b>6</b>
<b>4 Prerequisite.....</b>	<b>7</b>
1. Getting Started with Claude (Account Setup and Pro Upgrade).....	7
2. Set up Visual Studio Code Guide.....	7
3. Install & Setup Git, Node.js, Claude Code, Secrets & API Keys.....	7
<b>5 Cost To Perform This Lab.....</b>	<b>8</b>
<b>6 Setting Up the Lab Project.....</b>	<b>9</b>
<b>7 Pattern - 1 Task Decomposition.....</b>	<b>12</b>
7.1 The Decomposition Prompt.....	12
7.2 Hands-On: Decompose a Feature Request.....	13
<b>8 Pattern 2 - Planning.....</b>	<b>19</b>
8.1 Planning Prompts and Plan Mode.....	19
8.2 Hands-On: Generate and Review a Plan.....	19
<b>9 Pattern 3 - Implementation.....</b>	<b>24</b>
9.1 Controlled Implementation with Approval Gates.....	24
9.2 Hands-On: Implement the Feature Step by Step.....	24
<b>10 Pattern 4 - Review.....</b>	<b>28</b>
10.1 Self-Review vs Peer-Review Prompts.....	28
10.2 Hands-On: Review the Generated Code.....	29
<b>11 Pattern 5 - Verification.....</b>	<b>34</b>
11.1 Running Tests and Interpreting Results.....	34
11.2 Hands-On: Write and Run Verification Tests.....	34
<b>12 Putting It All Together - End-To-End Workflow.....</b>	<b>37</b>
<b>13 Share your learnings.....</b>	<b>38</b>
13.1 Share Hands-On Lab on LinkedIn to Build Authority & Job Opportunities.....	39
13.2 Share the win on the Community via Progress Diary.....	42
<b>14 Delete And Clean-Up.....</b>	<b>44</b>
<b>15 Troubleshooting.....</b>	<b>45</b>
<b>16 Summary.....</b>	<b>46</b>

# 1 INTRODUCTION

## Task Decomposition, Planning, Implementation, Review & Verification Patterns



When a junior developer receives a vague task like "build a password reset feature," the first thing they do is stare at the screen and hope something clicks. When a senior developer receives the same task, they break it into sub-tasks, sketch a plan, write the code, review their work, and run tests. This structured approach is what separates production engineers from beginners.

Claude Code is capable of doing both. By default it behaves like the junior — jumping straight to code. But when you prompt it using the right patterns, it behaves like the senior — decomposing, planning, implementing, reviewing, and verifying in that exact order.

In this lab, you will learn the five core patterns of agentic development and apply each one in hands-on exercises. By the end, you will have built and verified a small feature entirely through pattern-driven prompts — with no vibe-coding.

**In this lab, you will learn the five core patterns of agentic development and apply each one in hands-on exercises. Specifically, in this lab, we will cover:**

- **Project Setup:** Initializing a clean Node.js and Jest workspace tailored for AI collaboration.
- **The Decomposition Pattern:** Forcing Claude to break down a vague feature request (an email validator) into a structured list of atomic sub-tasks.
- **The Planning Pattern:** Generating and refining a strict, file-level execution plan before a single line of code is written.
- **The Implementation Pattern:** Executing the plan step-by-step using strict approval gates to maintain complete developer control.
- **The Review Pattern:** Leveraging Claude to perform rigorous self-reviews and peer reviews to catch edge cases, security risks, and logic gaps.
- **The Verification Pattern:** Guiding the AI to write, run, and pass an automated test suite to prove the implementation works definitively.

---

## 2 OVERVIEW & KEY CONCEPTS

In this section, we'll cover key concepts used in this guide, including an overview of what's covered in this guide.

---

### 2.1 Why Patterns Matter in Agentic

#### Development

An AI assistant without patterns produces code that looks right but fails in subtle ways — wrong edge cases, missing error handling, hallucinated APIs, untested logic. These failures cost hours to debug, especially when they surface in production.

Patterns fix this by forcing explicit intermediate outputs. Instead of prompting Claude with "build X" and hoping for the best, you prompt it through structured steps where each step produces something you can inspect before moving on:

- If decomposition misses a requirement, you catch it before any code is written.
- If the plan is flawed, you catch it before implementation.
- If the implementation has a bug, your review catches it before merge.
- If verification fails, you know exactly which assumption was wrong.

---

### 2.2 The Five Core Patterns

- **Decomposition:** Breaks a large user request into a list of smaller, independently solvable tasks. Used for any request that cannot fit in a single clear prompt.
- **Planning:** Orders the decomposed tasks, identifies dependencies, and produces an explicit plan. Claude confirms before acting. Used for high-stakes work where you want to review the approach BEFORE any code is written.
- **Implementation:** Executes each planned task in order, like writing code, creating files, and running commands. Used in every lab and production task.
- **Review:** Inspects the produced code against quality, correctness, security, and style standards. Used before commits and PRs, and for catching untested edge cases.

- **Verification:** Proves the implementation works, such as running tests, executing scripts, checks output against expected results. Used before calling any task complete. No code ships without execution proof.

---

## 2.3 How Claude Code Executes These Patterns

Claude Code has built-in mechanisms that support each pattern:

- **Plan Mode:** An explicit mode where Claude describes its plan before making any changes (enabled with `/plan` or settings).
- **Approval gates:** Before running commands or modifying files, Claude Code waits for your confirmation.
- **File diff previews:** Changes are shown as diffs before they are applied.
- **Integrated test execution:** Claude Code can run your test suite and interpret the output.
- **Context persistence:** Claude remembers the plan across implementation steps, staying on track without re-explaining.

### 3 DOCUMENTATION LINKS

1. Claude Code documentation:

<https://docs.claude.com/en/docs/claude-code>

2. Claude Code best practices:

<https://docs.claude.com/en/docs/claude-code/best-practices>

3. Anthropic's guide to agentic workflows:

<https://www.anthropic.com/engineering/claude-code-best-practices>

4. Claude Code plan mode reference:

<https://docs.claude.com/en/docs/claude-code/plan-mode>

K21Academy

## 4 PREREQUISITE

To complete this lab, you will need:

- Active Claude Pro subscription.
- VS Code installed with extensions.
- Claude Code CLI installed and logged in.
- Node.js 18+ and npm installed.
- Basic familiarity with JavaScript or Python.
- Approximately 60 minutes to complete the full lab end to end.

If you havent performed any of the pre-requisites for this lab then please refer to the below links:

1. [Getting Started with Claude \(Account Setup and Pro Upgrade\)](#)
2. [Set up Visual Studio Code Guide](#)
3. [Install & Setup Git, Node.js, Claude Code, Secrets & API Keys](#)

---

## 5 COST TO PERFORM THIS LAB

### Estimated Cost for This Lab:

**Total Estimated Cost:** \$0 additional (covered by your existing Claude Pro subscription). All tools used in this lab are free or included in your existing subscription. No new services are required.

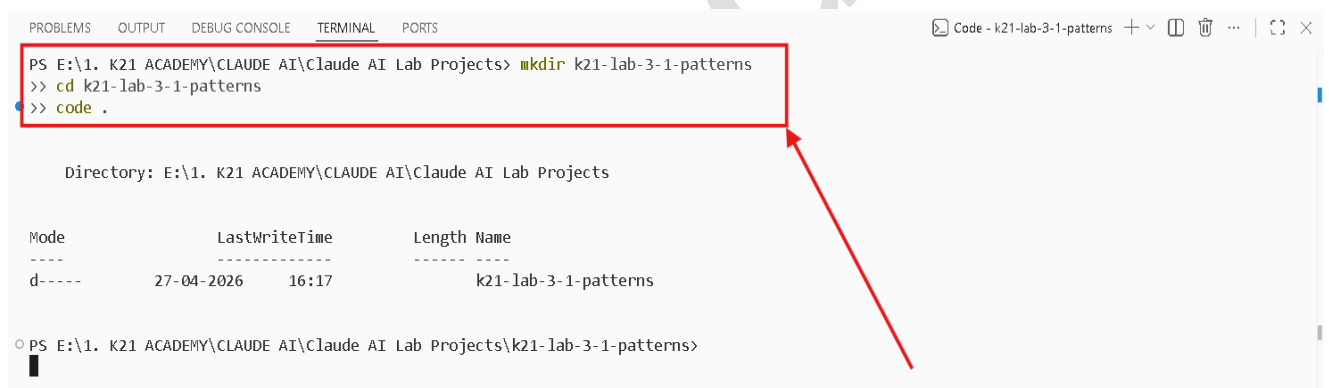
K21Academy

## 6 SETTING UP THE LAB PROJECT

Create a fresh project folder that will be used throughout this lab. This gives Claude Code a clean workspace to decompose, plan, implement, review, and verify.

1. Open VS Code.
2. Open the integrated terminal with `Ctrl + ``` (Cmd + `` on macOS).
3. Create a new project folder:

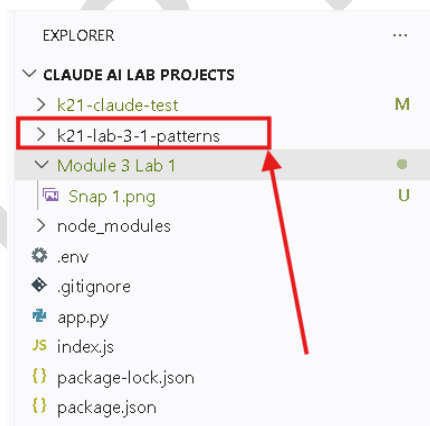
```
bash  
mkdir k21-lab-3-1-patterns  
cd k21-lab-3-1-patterns  
code .
```



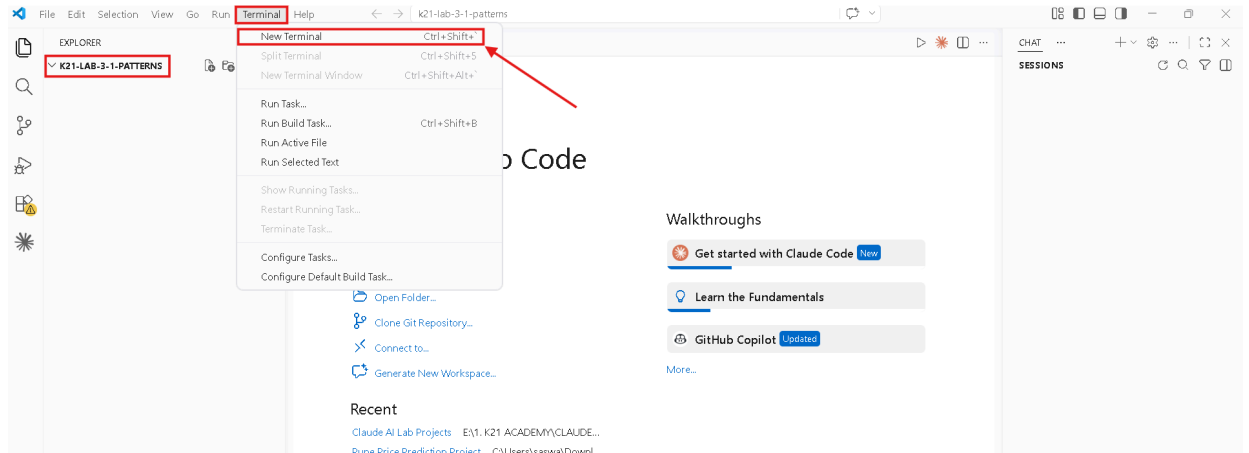
The screenshot shows the VS Code terminal interface. The terminal window is titled "Code - k21-lab-3-1-patterns". The command prompt shows the following sequence of commands: `PS E:\1. K21 ACADEMY\CLAUDE AI\Claude AI Lab Projects> mkdir k21-lab-3-1-patterns`, `>> cd k21-lab-3-1-patterns`, and `>> code .`. A red box highlights these commands. Below the terminal output, the directory listing for `E:\1. K21 ACADEMY\CLAUDE AI\Claude AI Lab Projects` is shown, with a table of files and folders. A red arrow points from the terminal output to the Explorer view in the next screenshot.

Mode	LastWriteTime	Length	Name
d----	27-04-2026 16:17		k21-lab-3-1-patterns

4. VS Code will reload with new folder open in the sidebar. Initialize a new [Node.js](#) project:



```
bash
npm init -y
```



```
PS E:\1. K21 ACADEMY\CLAUDE AI\Claude AI Lab Projects\k21-lab-3-1-patterns> npm init -y
Wrote to E:\1. K21 ACADEMY\CLAUDE AI\Claude AI Lab Projects\k21-lab-3-1-patterns\package.json:

{
  "name": "k21-lab-3-1-patterns",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "type": "commonjs"
}
```

5. Install a test framework (we will use Jest for the verification pattern):

```
bash
npm install --save-dev jest
```

```
PS E:\1. K21 ACADEMY\CLAUDE AI\Claude AI Lab Projects\k21-lab-3-1-patterns> npm install --save-dev jest
npm warn deprecated inflight@1.0.6: This module is not supported, and leaks memory. Do not use it. Check out lru-cache if you want a good and tested way to coalesce async requests by a key value, which is much more comprehensive and powerful.
npm warn deprecated glob@10.5.0: Old versions of glob are not supported, and contain widely publicized security vulnerabilities, which have been fixed in the current version. Please update. Support for old versions may be purchased (at exorbitant rates) by contacting i@izs.me
npm warn deprecated glob@7.2.3: Old versions of glob are not supported, and contain widely publicized security vulnerabilities, which have been fixed in the current version. Please update. Support for old versions may be purchased (at exorbitant rates) by contacting i@izs.me

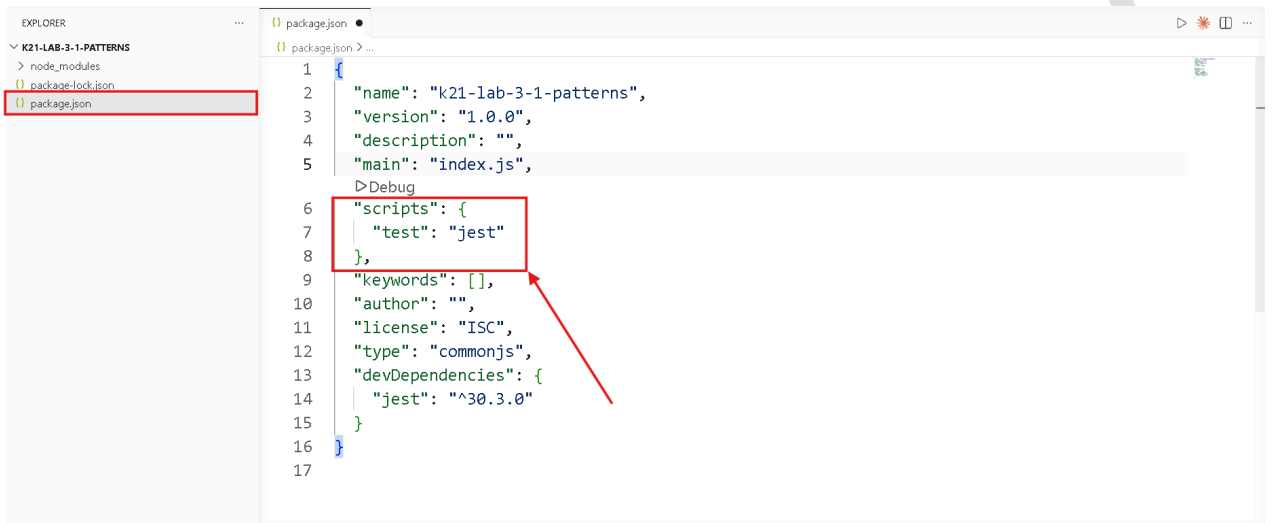
added 293 packages, and audited 294 packages in 27s

44 packages are looking for funding
  run 'npm fund' for details

found 0 vulnerabilities
PS E:\1. K21 ACADEMY\CLAUDE AI\Claude AI Lab Projects\k21-lab-3-1-patterns>
```

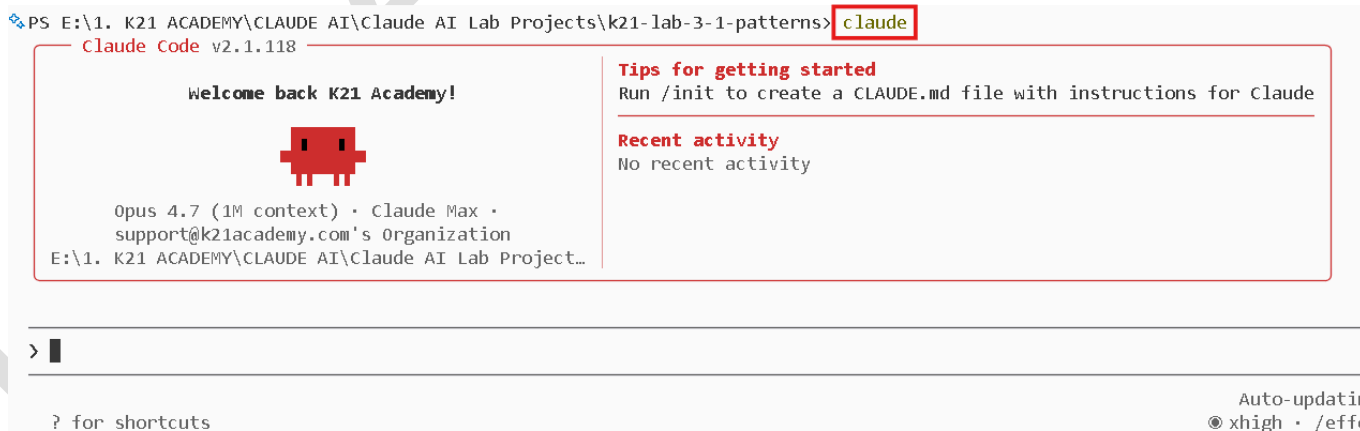
6. Open the `package.json` file and update the scripts section:

```
json  
"scripts": {  
  "test": "jest"  
}
```



7. Start the Claude Code session in the project root:

```
bash  
claude
```



You are now ready to apply the five patterns. Keep the Claude Code session open for the remainder of the lab.

## 7 PATTERN - 1 TASK DECOMPOSITION

Decomposition turns a vague, high-level request into a list of concrete, individually solvable sub-tasks. This is the single best leverage point in agentic development.

### 7.1 The Decomposition Prompt

A good decomposition prompt does three things: (1) asks Claude NOT to write code yet, (2) demands a numbered list of sub-tasks, (3) asks for any clarifying questions.

#### Template Prompt:

Do NOT write any code yet. I want you to decompose the following feature into numbered sub-tasks:

Feature: <describe the feature in one or two sentences>

For each sub-task, give me:

1. A short title
2. One-sentence description
3. Any external dependencies (libraries, APIs, files)
4. Approximate difficulty (trivial / medium/complex)

If anything is ambiguous, list your clarifying questions at the end.

```
> Do NOT write any code yet. I want you to decompose the following feature into numbered sub-tasks:  
Feature: <describe the feature in one or two sentences>  
  
For each sub-task, give me:  
1. A short title  
2. One-sentence description  
3. Any external dependencies (libraries, APIs, files)  
4. Approximate difficulty (trivial / medium / complex)  
  
If anything is ambiguous, list your clarifying questions at the end.
```

• The feature description in your prompt is still a placeholder – `<describe the feature in one or two sentences>` – so I don't know what to decompose.

Could you tell me what feature you want broken down? A couple of examples of the level of detail that works well:

- "Add Google sign-in to our Next.js app so users can authenticate with their Google account instead of email/password."
- "Build a CSV export button on the orders dashboard that downloads the currently filtered rows."
- "Add a daily cron job that emails admins a summary of new signups from the last 24 hours."

A few extra hints that will sharpen the decomposition (optional, but helpful):

1. **Stack/context** – is this for the `k21-lab-3-1-patterns` project I can see in your working directory, or something else? (I notice `package.json` is open in your IDE – want me to peek at it to ground the breakdown in the actual stack?)
2. **Scope** – MVP vs. production-ready (tests, error handling, observability)?
3. **Constraints** – any libraries, APIs, or services it must use or avoid?

Once you fill in the feature sentence, I'll return numbered sub-tasks in the exact format you asked for (title, one-line description, external dependencies, difficulty), plus a clarifying-questions list at the end if anything's still ambiguous.

## 7.2 Hands-On: Decompose a Feature Request

You will build a small utility — an email validator function that detects common disposable email domains and returns a structured result.

1. In your Claude Code session, paste the following prompt:

```
Do NOT write any code yet. I want you to decompose the following feature into numbered sub-tasks:
```

```
Feature: Build a JavaScript module that validates email addresses. It must check standard email format, detect disposable email providers (e.g., mailinator, 10minutemail), and return a result object with `valid`, `reason`, and `isDisposable` fields.
```

```
For each sub-task, give me:
```

1. A short title
2. One-sentence description
3. Any external dependencies (libraries, APIs, files)

#### 4. Approximate difficulty (trivial / medium / complex)

If anything is ambiguous, list your clarifying questions at the end.

```
> Do NOT write any code yet. I want you to decompose the following feature into numbered sub-tasks:

Feature: Build a JavaScript module that validates email addresses. It must check standard email format, detect disposable email providers (e.g., mailinator, 10minutemail), and return a result object with `valid`, `reason`, and `isDisposable` fields.

For each sub-task, give me:
1. A short title
2. One-sentence description
3. Any external dependencies (libraries, APIs, files)
4. Approximate difficulty (trivial / medium / complex)

If anything is ambiguous, list your clarifying questions at the end.
```

2. Review Claude's decomposition. You should see sub-tasks like: validate email format, maintain a list of disposable domains, parse the email, and build the result object.

#### • *Sub-task decomposition*

##### 1. Module scaffolding & public API shape

- **Description:** Set up the file (e.g., `validateEmail.js`), define the exported function signature, and lock down the result-object contract `{ valid, reason, isDisposable }`.
- **Dependencies:** None (uses existing `package.json` – currently open in your IDE – to decide CommonJS vs. ESM).
- **Difficulty:** trivial

##### 2. Syntactic format validation

- **Description:** Validate the email string against a standard format (RFC 5322-pragmatic regex or a library) and produce a clear failure `reason` when it fails.
- **Dependencies:** Optional – `validator` npm package, or a hand-rolled regex (no dep).
- **Difficulty:** medium (*trivial if using a library, medium if hand-rolled – regex edge cases bite*)

##### 3. Domain extraction utility

- **Description:** Helper that splits the local-part from the domain and lowercases/normalizes the domain for downstream checks.
- **Dependencies:** None.
- **Difficulty:** trivial

##### 4. Disposable-provider data source

- **Description:** Decide where the blacklist lives – bundled JSON file, npm package like `disposable-email-domains`, or a remote API – and load it into the module.
- **Dependencies:** Either `disposable-email-domains` (npm), a local `disposable-domains.json`, or an external API (e.g., Kickbox/Debounce). Pick one.
- **Difficulty:** medium (*decision-heavy; affects freshness, bundle size, offline behavior*)

##### 5. Disposable-provider detection

- **Description:** Match the extracted domain (and optionally subdomains) against the blacklist and set `isDisposable`.
- **Dependencies:** Output of sub-task 4.
- **Difficulty:** trivial (*if exact-match*) / medium (*if subdomain or wildcard matching is required*)

##### 6. Result assembly & reason taxonomy

- **Description:** Define the canonical set of `reason` strings (e.g., "ok", "invalid\_format", "disposable\_provider", "empty\_input") and assemble the final result object consistently.
- **Dependencies:** None.
- **Difficulty:** trivial

#### 7. Input guards & edge cases

- **Description:** Handle non-string input, empty strings, leading/trailing whitespace, and very long inputs without throwing.
- **Dependencies:** None.
- **Difficulty:** trivial

#### 8. Unit tests

- **Description:** Cover valid emails, malformed emails, known disposable domains, edge cases (Unicode, plus-addressing, subdomains), and the result-object shape.
- **Dependencies:** A test runner – Jest, Vitest, or Node's built-in `node:test`.
- **Difficulty:** medium

#### 9. Documentation / usage example

- **Description:** Short README section or JSDoc block showing import, call, and example return values for each `reason`.
- **Dependencies:** None.
- **Difficulty:** trivial

#### 10. (Optional) MX-record / deliverability check

- **Description:** Resolve the domain's MX records to flag domains that can't actually receive mail.
- **Dependencies:** Node's `dns/promises` (Node-only – won't work in browsers).
- **Difficulty:** complex (*async, network-dependent, environment-sensitive*)

3. If Claude listed clarifying questions, answer them now in the same session. This tightens the decomposition.

#### Clarifying questions

1. **Runtime target** – Node.js only, browser-compatible, or both? This decides whether DNS/MX checks are even on the table and constrains the disposable-list approach.
2. **Module format** – CommonJS, ESM, or dual? (Want me to read your `package.json` to infer this?)
3. **Disposable list source** – bundled static list (fast, stale over time), npm package (decent middle ground), or live API (always fresh, network-dependent)?
4. **Sync vs. async API** – synchronous return (no MX/API checks) or `Promise<Result>` (allows future network checks)?
5. **reason field semantics** – single primary reason, or an array when multiple things are wrong (e.g., malformed *and* disposable)? Right now the spec implies a single string.
6. **Should a disposable-but-syntactically-valid email return `valid: true` or `valid: false`?** Spec is ambiguous – `isDisposable` is a separate field, but the caller needs to know whether `valid` is "format-only valid" or "format-valid AND not disposable."
7. **Scope of "standard email format"** – strict RFC 5322 (allows quirky-but-legal addresses) or pragmatic/common-case (most apps prefer this)?
8. **Allow-list / overrides** – should callers be able to whitelist specific disposable domains or extend the blocklist?
9. **Internationalized domains (IDN/Unicode)** – must we support Punycode / non-ASCII domains, or is ASCII-only fine for v1?

Once you answer #1, #3, #4, and #6, I have enough to start implementing.

4. Now, based on what the rest of your Lab 3.1 guide expects (specifically looking ahead to the Planning and Verification steps), here is exactly how you should answer those four questions.
5. Copy and paste this text directly into your Claude Code terminal and hit Enter:

Here are the answers to your questions:

1. Runtime target: Node.js only.
3. Disposable list source: A bundled static list (we will just use a local data file for the domains).
4. Sync vs. async API: Synchronous return.
6. Valid logic: A disposable-but-syntactically-valid email should return ``valid: true`` and ``isDisposable: true``.

```
> Here are the answers to your questions:
1. Runtime target: Node.js only.
3. Disposable list source: A bundled static list (we will just use a local data file for the domains).
4. Sync vs. async API: Synchronous return.
6. Valid logic: A disposable-but-syntactically-valid email should return `valid: true` and `isDisposable: true`.
```

- Let me peek at your `package.json` so I can lock in the module format without asking.

Read 1 file (ctrl+o to expand)

- Good – that resolves two open items: **CommonJS** and **Jest** are already set up. Here's the locked-in plan based on your answers.

---

Locked-in decisions

Question	Answer
Runtime	Node.js only
Module format	CommonJS ( <code>module.exports</code> ) – from <code>package.json</code>
Disposable list source	Bundled local data file
API style	Synchronous
Disposable + valid format	<code>valid: true, isDisposable: true</code>
Test runner	Jest 30 – already in <code>devDependencies</code>
MX/DNS check	<b>Dropped</b> (sync-only, out of scope)

---

Updated sub-task list

**1. Module scaffolding & exports**

- **Description:** Create the validator file, define the function signature, and wire up `module.exports` for CommonJS.
- **Dependencies:** None.
- **Difficulty:** trivial

**2. Bundled disposable-domains data file**

- **Description:** Create a local `disposable-domains.json` (or `.js`) seeded with common providers (mailinator, 10minutemail, guerrillamail, tempmail, yopmail, etc.) and `require` it from the validator.



### 3. Syntactic format validation

- **Description:** Validate the email string against a pragmatic regex and return a clear failure `reason` when malformed.
- **Dependencies:** None (hand-rolled regex; no external lib needed for v1).
- **Difficulty:** medium (*regex edge cases – quoted local-parts, consecutive dots, trailing dots*)

### 4. Domain extraction & normalization

- **Description:** Helper that splits on @, lowercases the domain, and trims whitespace before the blacklist check.
- **Dependencies:** None.
- **Difficulty:** trivial

### 5. Disposable-provider detection

- **Description:** Exact-match the normalized domain against the bundled list and set `isDisposable` accordingly.
- **Dependencies:** Output of sub-task 2.
- **Difficulty:** trivial

### 6. Result assembly & `reason` taxonomy

- **Description:** Define canonical `reason` values (`"ok"`, `"invalid_format"`, `"empty_input"`, `"not_a_string"`) and assemble the result object – given your rule, a disposable-but-valid email returns `{ valid: true, reason: "ok", isDisposable: true }`.
- **Dependencies:** None.
- **Difficulty:** trivial

### 7. Input guards

- **Description:** Handle non-string inputs, empty strings, and surrounding whitespace without throwing.
- **Dependencies:** None.
- **Difficulty:** trivial

### 8. Jest unit tests

- **Description:** Cover valid emails, malformed emails, disposable hits, non-string input, whitespace, and verify the exact result-object shape for each branch.
- **Dependencies:** Jest 30 (*already in devDependencies*).
- **Difficulty:** medium

---

#### Two small things still ambiguous

1. **Subdomain matching** – should `foo@mail.mailinator.com` count as disposable, or only an exact domain match against `mailinator.com`? (Most real-world blocklists do subdomain matching; it's one extra line.)
2. **reason for the disposable-but-valid case** – confirming you want `reason: "ok"` (since `valid: true`), with `isDisposable: true` as the sole signal. The alternative would be `reason: "disposable_provider"` even when `valid: true`, but that's a weirder contract.

If you confirm those two, I'll have everything I need to implement. Want me to proceed to code after you answer, or hold for further planning?

\*Baked for 37s

※ **recap:** Goal: build a sync CommonJS email validator with format check, bundled disposable-domain list, and a `{valid, reason, isDisposable}` result. Next: confirm subdomain matching behavior and the `reason` value for disposable-but-valid emails, then implement. (disable recaps in `/config`)

## 6. Why we are choosing these answers (for your own learning):

- **#1 & #4:** Keeping it "Node.js only" and "Synchronous" keeps the lab simple and matches the Jest testing setup you did in Step 6.
- **#3:** In Step 8.2 of your lab, the plan expects Claude to create a specific file called `src/disposableDomains.js`. Telling it to use a "bundled static list" ensures it creates that exact file instead of trying to download a random NPM package.
- **#6:** In Step 11.1 of your lab, the test requirements specifically state that a mailinator email must return `{ valid: true, isDisposable: true }`.

7. Once you paste those answers in, Claude will give you the final, clean decomposition list, and you can move right on to **Step 8 (Pattern 2 — Planning)**!

**Note:** A strong decomposition is atomic, independently testable, ordered by dependency, and explicit about ambiguity. If Claude gives a bad decomposition, reply: "Redo the decomposition — each sub-task must be atomic and independently testable."

K21Academy

## 8 PATTERN 2 - PLANNING

Planning takes the decomposed sub-tasks and adds order, dependencies, file layout, and a specific execution strategy.

### 8.1 Planning Prompts and Plan Mode

You can drive planning through prompts or use Claude's built-in Plan mode (`/plan`).

**Template prompt (prompt-driven planning):**

Now turn the decomposition above into a full implementation plan. Include:

1. The exact files you will create and their paths
2. The public function signatures (name, parameters, return type)
3. The test file names and what each test will cover
4. The order in which you will implement each file
5. Any pattern or library decisions (e.g., using a regex vs a validation library)

Do NOT write any code yet. I need to review and approve the plan first.

```
> Now turn the decomposition above into a full implementation plan. Include:  
  
1. The exact files you will create and their paths  
2. The public function signatures (name, parameters, return type)  
3. The test file names and what each test will cover  
4. The order in which you will implement each file  
5. Any pattern or library decisions (e.g., using a regex vs a validation library)  
  
Do NOT write any code yet. I need to review and approve the plan first.
```

### 8.2 Hands-On: Generate and Review a Plan

1. In your Claude Code session, past the planning prompt above.
2. Wait for Claude to produce plan. A good plan will look something like:
  - Files : `src/validator.js`, `src/disposableDomains.js`,  
`tests/validator.test.js`

- **Signatures:** `validateEmail(email: string)`
- **Order:** 1) disposableDomains.js, 2) validator.js, 3) tests

• *Implementation Plan*

**Pre-plan defaults for the two open questions**

I'll propose defaults so the plan is concrete – flag any you want changed before I start.

Open question	Proposed default	Rationale
Subdomain matching	<b>Yes</b> – <code>foo@mail.mailinator.com</code> counts as disposable	Real blocklists work this way; trivial extra logic
<code>reason</code> when valid + disposable	<b>"ok"</b>	<code>valid: true</code> ⇒ no failure to report; <code>isDisposable</code> is the dedicated signal

---  
**1. Files to create**

All paths are relative to the project root (`e:\1. K21 ACADEMY\CLAUDE AI\Claude AI Lab Projects\k21-lab-3-1-patterns\`). Flat structure – appropriate for a small lab module.

Path	Purpose
<code>validateEmail.js</code>	Public module: <code>validateEmail</code> function + private helpers
<code>disposable-domains.json</code>	Bundled blocklist (array of lowercase domain strings)
<code>validateEmail.test.js</code>	Jest test suite

**Notes:**

- I will **not** modify `index.js` or `package.json` unless you ask.
- No new `devDependencies` needed – Jest 30 is already installed.
- No new `dependencies` – zero-runtime-dep module.



## 2. Public function signature

File: `validateEmail.js`

```
validateEmail(input)
```

<b>Name</b>	<code>validateEmail</code>
<b>Parameters</b>	<code>input</code> – expected to be a <code>string</code> ; any other type is handled, not thrown
<b>Returns</b>	<code>ValidationResult</code> object (synchronous)
<b>Throws</b>	Never – invalid inputs return a structured result
<b>Export style</b>	<code>module.exports = validateEmail;</code> (single default export, CommonJS)

**ValidationResult** shape (JSDoc-typed):

```
{
  valid:      boolean // true = syntactically valid email
  reason:     string   // one of the enum values below
  isDisposable: boolean // true = domain matches blacklist (only meaningful when valid===true)
}
```

**reason** enum (closed set):

- "ok" – passed all syntactic checks
- "empty\_input" – empty string or whitespace-only
- "not\_a\_string" – input was `null`, `undefined`, `number`, `object`, etc.
- "invalid\_format" – failed the regex check

When `valid === false`, `isDisposable` is always `false` (we don't probe the blacklist for malformed input).

**Private helpers (not exported):**

- `extractDomain(email: string): string` – splits on the *last* `@`, lowercases, trims
- `isDisposableDomain(domain: string): boolean` – exact match OR subdomain match against the bundled list



### 3. Test file & coverage

File: `validateEmail.test.js`

Organized with `describe` blocks. Every test asserts the **full result object shape**, not just one field – this catches accidental contract drift.

#### `describe("input guards")`

1. returns `{ valid: false, reason: "not_a_string", isDisposable: false }` for undefined
2. ...for null
3. ...for a number (42)
4. ...for an object (`{}`)
5. returns `{ valid: false, reason: "empty_input", isDisposable: false }` for `""`
6. ...for `" "` (whitespace only)

#### `describe("format validation - invalid")`

7. `"plainstring"` → `invalid_format`
8. `"missing@tld"` → `invalid_format` (no dot in domain)
9. `"@nodomain.com"` → `invalid_format` (empty local-part)
10. `"spaces in@local.com"` → `invalid_format`
11. `"double@@at.com"` → `invalid_format`
12. `"trailing.dot.@example.com"` → `invalid_format`

#### `describe("format validation - valid, non-disposable")`

13. `"alice@example.com"` → `{ valid: true, reason: "ok", isDisposable: false }`
14. `"alice+tag@example.com"` (plus addressing) → valid, non-disposable
15. `"a.b.c@sub.example.co.uk"` (multi-label domain) → valid, non-disposable
16. `" alice@example.com "` (surrounding whitespace) → valid, non-disposable

#### `describe("disposable detection")`

17. `"foo@mailinator.com"` → `{ valid: true, reason: "ok", isDisposable: true }`
18. `"foo@10minutemail.com"` → disposable
19. `"foo@MAILINATOR.COM"` (uppercase) → disposable (case-insensitive)
20. `"foo@mail.mailinator.com"` (subdomain) → disposable (*per default in pre-plan table*)
21. `"foo@gmail.com"` → not disposable (Gmail must never be flagged – sanity check)

#### `describe("contract invariants")`

22. Every result has exactly the three keys `valid`, `reason`, `isDisposable` – no extras
23. `reason` is always one of the four enum values
24. `isDisposable` is always `false` when `valid` is `false`

**Total: 24 tests across 5 describe blocks.**

---

### 4. Implementation order

I'll work in this order so each step has the inputs it needs and the test suite stays green from step 3 onward:

1. **disposable-domains.json** – seed list (`mailinator.com`, `10minutemail.com`, `guerrillamail.com`, `yopmail.com`, `tempmail.com`, `throwawaymail.com`, `getnada.com`, `dispostable.com`, `trashmail.com`, `fakeinbox.com` – ~10 entries to start). Data first because step 2 will `require()` it.
2. **validateEmail.js skeleton** – exports a stub that always returns `{ valid: false, reason: "not_a_string", isDisposable: false }`. Verifies the wiring before any logic.
3. **validateEmail.test.js** – write the full suite from §3 against the stub. Most tests fail; a few input-guard tests pass. Establishes the contract.
4. **Input guards** in `validateEmail.js` – handles `null/undefined/non-string/empty/whitespace`. Tests 1-6 go green.
5. **Format regex + extractDomain helper** – tests 7-16 go green.
6. **isDisposableDomain helper + final result wiring** – tests 17-21 go green.
7. **JSDoc block** on `validateEmail` documenting the parameter, return shape, and `reason` enum. Run the full suite once more – all 24 should pass.

Each step ends with `npm test` to confirm.

### 5. Pattern & library decisions

Decision	Choice	Reason
Validation approach	<b>Hand-rolled regex</b> , not <code>validator</code> npm package	Zero deps fits a lab/learning project; regex coverage is sufficient for the spec ("standard email format")
Regex flavor	<b>Pragmatic</b> - <code>/^[^\s@]+@[^\s@]+\.[^\s@]+\$/</code> as the base, hardened against <code>..</code> , leading/trailing dots, and <code>@@</code>	Strict RFC 5322 is overkill and rejects emails real apps accept
Blocklist storage	<b>.json file</b> , not <code>.js</code>	Easier to update by hand or by script; cleanly <code>require</code> -able in CommonJS
Blocklist matching	<b>Set-based exact match + subdomain check</b> via <code>endsWith(".") + domain</code>	O(1) exact match; subdomain check is a single extra comparison per blocklist entry - fine for a 10-100 entry list
Module shape	<b>Single function export</b> , helpers kept private inside the file	Smallest possible API surface; helpers are tested through the public function
Error handling	<b>No throws</b> - all bad input becomes a structured result	Makes the function safe to call without <code>try/catch</code>
Async	<b>Synchronous only</b>	Matches your spec; no DNS/MX
Code comments	Minimal - only a JSDoc block on the public function	Per project norms; identifiers should be self-explanatory
File layout	<b>Flat</b> (no <code>src/</code> , no <code>__tests__/</code> )	Module is three files; folder structure would be ceremony

---  
what I need from you to proceed

---  
what I need from you to proceed

Confirm or override:

1. **Subdomain matching** = yes  / change to `exact-match-only`?
2. **reason: "ok" for valid-but-disposable**  / change to `"disposable_provider"`?
3. **Flat file layout at project root**  / put files under `src/` and `__tests__/`?
4. **Initial blocklist seed of ~10 well-known providers**  / want a different seed list?

Once you approve, I'll implement in the order in §4.

3. Review the plan critically. Are the file paths sensible? Is the order correct?
4. Iterate until the plan is exactly what you want. Only then approve with: *"Plan approved. Proceed to implementation."*

**Note:** Treat the plan like a contract. If Claude deviates mid-implementation, pull it back with: "Stop. The plan specified X. Why did you do Y?"

## 9 PATTERN 3 - IMPLEMENTATION

Implementation is where the plan becomes actual code. The goal is controlled, step-by-step execution with approval gates.

### 9.1 Controlled Implementation with Approval

#### Gates

Claude Code, by default, asks for approval before creating or modifying files. Keep this default ON.

#### Implementation prompt after plan approval:

Plan approved. The defaults for 1, 2, and 4 are perfect. For 3, put the files under src/ and tests/.

Now, implement the plan one file at a time, starting with the first file in the implementation order.

After each file, PAUSE and wait for my explicit approval before moving to the next file. Do not proceed until I confirm.

For each file, include inline comments explaining any non-obvious decisions.

### 9.2 Hands-On: Implement the Feature Step by Step

1. In your Claude Code session, paste the implementation prompt above.

```
> Plan approved. The defaults for 1, 2, and 4 are perfect. For 3, put the files under src/ and tests/.  
  
Now, implement the plan one file at a time, starting with the first file in the implementation order.  
  
After each file, PAUSE and wait for my explicit approval before moving to the next file. Do not proceed until I confirm.  
  
For each file, include inline comments explaining any non-obvious decisions.
```

```
+ Germinating.. (2s · thinking)  
L Tip: Hit Enter to queue up additional messages while Claude is working.
```

2. Claude will create the first file. Approve the file creation when prompted.

```
Opened changes in Visual Studio Code

Save file to continue...

Do you want to make this edit to disposable-domains.json?
> 1. Yes
  2. Yes, allow all edits during this session (shift+tab)
  3. No

Esc to cancel · Tab to amend
```

3. Press **Enter** on your keyboard to select "Yes". (This saves the file and closes the side-by-side view).

```
? for shortcuts
 6 "guerrillamail.com",
 7 "mailinator.com",
 8 "tempmail.com",
 9 "throwawaymail.com",
10 "trashmail.com",
... +2 lines (ctrl+o to expand)

• File 1 of 3 complete. src/disposable-domains.json created.

Notes on this file

- No inline comments in the file itself – strict JSON forbids comments. Surfacing the non-obvious decisions here instead:
- Sorted alphabetically. Keeps diffs minimal when entries are added later (10minutemail.com sorts first because digits precede letters in ASCII).
- All lowercase. isDisposableDomain will normalize the input domain to lowercase before lookup, so the list must already be lowercase to match.
- JSON, not JS module. Easier for a script or a non-JS tool to update later; cleanly require-able from CommonJS.
- Ten well-known providers as the starter set. Enough breadth to write meaningful tests (including the subdomain case via mailinator.com), small enough to be obviously curated rather than exhaustive – production projects would swap this for the disposable-email-domains package or a remote feed.

Revised plan note (one small reorder)

Your "one file at a time, pause after each" instruction is cleaner if I write validateEmail.js fully (logic + JSDoc) before the test file, instead of the staged "skeleton → tests → fill in" sequence in my original §4. Same end state, fewer half-built files for you to review. Flagging in case you want me to stick to the original staged order instead.

Next up (awaiting approval)

File 2 of 3: src/validateEmail.js – full implementation (input guards, regex, domain extraction, disposable detection, result assembly, JSDoc).

Approve to proceed?

*Crunched for 1m 10s

> █
```

4. Reply to Claude with: *"Approved. Continue to next file."*

Approve to proceed?

\*Crunched for 1m 10s

> Approved. Continue to next file. █

## 5. Repeat for each file:

Opened changes in Visual Studio Code

Save file to continue...

Do you want to make this edit to `validateEmail.js`?

- > 1. Yes
- 2. Yes, allow all edits during this session (**shift+tab**)
- 3. No

Esc to cancel · Tab to amend

Next up (awaiting approval)

File 3 of 3: `tests/validateEmail.test.js` – 24 Jest tests across 5 `describe` blocks, exactly per §3 of the approved plan.

Approve to proceed?

\*Worked for 1m 39s

> Approved. Continue to next file. █

## 6. Repeat Again

Opened changes in Visual Studio Code

Save file to continue...

Do you want to make this edit to `validateEmail.test.js`?

- > 1. Yes
- 2. Yes, allow all edits during this session (**shift+tab**)
- 3. No

Esc to cancel · Tab to amend

## 7. You have now completed **Step 9 (Pattern 3 — Implementation)**

Copyright© K21Academy | All Rights Reserved

Sharing, Reselling, or duplication of this content is strictly prohibited without K21Academy's written permission

• **File 3 of 3 complete.** All 24 tests pass.

Test Suites: 1 passed, 1 total  
Tests: 24 passed, 24 total

**Notes on this file**

- **toEqual** against the full result object instead of per-field asserts – explained in the file's top comment. A field-by-field check (`expect(result.valid).toBe(true)`) would silently miss a stray extra key or a typo in `isDisposable`.
- **SAMPLE\_INPUTS** shared by the three invariant tests. When a new `reason` branch gets added later, the only change needed is extending that one array – the three invariants automatically cover it.
- **Test 24 uses an if guard** (only asserts when `valid === false`) rather than filtering the sample first. Reads more directly as "for every input, if it's invalid, then `isDisposable` must be false" – which is exactly the contract.

**Implementation summary**

File	Status
<code>src/disposable-domains.json</code>	✔ 10 seed providers
<code>src/validateEmail.js</code>	✔ Full module, JSDoc-typed
<code>tests/validateEmail.test.js</code>	✔ 24/24 passing

The module is ready to use: `const validateEmail = require('./src/validateEmail');`. Nothing else was modified – `package.json`, `index.js`, and `node_modules` are untouched.

\* Sautéed for 1m 35s

※ **recap:** Built a JavaScript email validator module (format check + disposable-provider detection) under `src/` and `tests/`, with all 24 Jest tests passing. Module is complete and ready to use; awaiting your next instruction. (disable recaps in `/config`)

8. Now verify the folder structure in you terminal by opening a new **bash** terminal:

```
saswa@ImmEthicalHkr MINGW64 /e/1. K21 ACADEMY/CLAUDE AI/Claude AI Lab Projects/k21-lab-3-1-patterns (main)
$ ls -la src/ tests/
src/:
total 5
drwxr-xr-x 1 saswa 197609  0 Apr 28 15:51 ./
drwxr-xr-x 1 saswa 197609  0 Apr 28 16:02 ../
-rw-r--r-- 1 saswa 197609 202 Apr 28 14:28 disposable-domains.json
-rw-r--r-- 1 saswa 197609 2661 Apr 28 15:51 validateEmail.js

tests/:
total 8
drwxr-xr-x 1 saswa 197609  0 Apr 28 16:02 ./
drwxr-xr-x 1 saswa 197609  0 Apr 28 16:02 ../
-rw-r--r-- 1 saswa 197609 5657 Apr 28 16:02 validateEmail.test.js
```

This will list out the files inside those folders without causing an error.

Once you verify the files are there, you can go back to your Claude session

## 10 PATTERN 4 - REVIEW

Review evaluates the whole implementation as a unit for error handling, failure modes, and security holes.

### 10.1 Self-Review vs Peer-Review Prompts

#### Self-Review prompt:

Now step back and review the entire implementation you just produced. Look at it as if you were a senior engineer doing code review. Specifically check:

1. Error handling — are all error paths covered?
2. Edge cases — empty strings, null, very long inputs, Unicode?
3. Security — any injection risks, unsafe regex (ReDoS), information leaks?
4. Readability — are function names and variables self-explanatory?
5. Test coverage — any code paths not exercised by tests?

List issues found, ordered by severity. Propose a fix for each. Do not change code yet.

#### Peer-review Prompt:

Review this module from the perspective of a senior backend engineer who has been bitten by production bugs. Focus on:

1. What happens if this runs on 10,000 emails per second?
2. What happens if the disposable domain list doubles in size?
3. What happens if a user passes malformed Unicode?
4. What happens if a regex takes longer than expected?

List real issues, not theoretical ones. Suggest fixes.

## 10.2 Hands-On: Review the Generated Code

1. In your session, run the self-review prompt.

```
> Now step back and review the entire implementation you just produced. Look at it as if you were a senior engineer doing code review. Specifically check:  
  
1. Error handling – are all error paths covered?  
2. Edge cases – empty strings, null, very long inputs, Unicode?  
3. Security – any injection risks, unsafe regex (ReDoS), information leaks?  
4. Readability – are function names and variables self-explanatory?  
5. Test coverage – any code paths not exercised by tests?  
  
List issues found, ordered by severity. Propose a fix for each. Do not change code yet.
```

2. Read the output. Claude will usually find 2-4 real issues.

K21Academy

• **Code Review: validateEmail module**

Reviewing the three files I just shipped, looking for issues I missed during implementation. Ordered by severity.

---

**MEDIUM severity**

**M1. No length cap → DoS via large inputs**

**Where:** `src/validateEmail.js:39` – regex runs on whatever string the caller passes.

The regex is safe from *catastrophic* ReDoS (I traced the backtracking – all quantifiers are linear), but a 10MB string still costs  $O(n)$  CPU through `trim()` and `EMAIL_REGEX.test()`. RFC 5321 caps email at 254 chars; anything beyond ~320 is obviously invalid and shouldn't waste cycles.

**Fix:** After the type check, before `trim()`, reject inputs over a sane upper bound:

```
if (input.length > 320) {  
  return { valid: false, reason: 'invalid_format', isDisposable: false };  
}
```

Or add a new `reason: 'too_long'` to the enum if callers need to distinguish. Note this requires updating §3 of the plan and bumping the test count.

**M2. Documented regex guards have no tests**

**Where:** `src/validateEmail.js:13-14` claims the regex rejects consecutive dots (`a..b@x.com`) and leading dot in local part (`.foo@x.com`), but `tests/validateEmail.test.js` covers neither.

A future regex tweak could silently break these guards – the comment would become a lie and no test would fail. The current 24 tests don't actually exercise the `(?!\\.)` and `(?!.*\\.\\.)` lookaheads.

**Fix:** Add two tests in the `format validation – invalid` block:

```
- expect(validateEmail('a..b@example.com')) → invalid_format  
- expect(validateEmail('.foo@example.com')) → invalid_format
```

**M3. Unicode / IDN behavior is undefined**

**where:** `src/validateEmail.js:18` – `[^\s@]+` accepts any non-whitespace, non-@ character, including non-ASCII.

- `用户@example.com` passes the regex. Whether that's correct depends on whether you want EAI (RFC 6531) support.  
- Worse: the disposable list is ASCII-only and stored as Punycode-or-Unicode is unspecified. A user with `foo@xn--mailinator-...` (hypothetical Punycode form) wouldn't match a Unicode entry, and vice versa. No homograph normalization either.

**Fix:** Pick one:

1. **Document it as ASCII-only:** tighten the regex to `[A-Za-z0-9._%+-]` for local and `[A-Za-z0-9.-]` for domain – explicit, predictable, and covers ~all real-world emails.
2. **Embrace Unicode:** keep current regex, but normalize the domain via `new URL('http://' + domain).hostname` (which lowercases and Punycode-encodes), then match against a Punycode-canonicalized blacklist.

Recommend option 1 for v1 – it matches the "pragmatic, not strict RFC" stance already taken.

---

**LOW severity**

**L1. Control characters slip through**

**where:** Same regex.

`foo\x00@example.com` and `foo\x01@example.com` pass (null bytes and other C0 control chars aren't `\s` and aren't `@`). Not a vulnerability in *this* module – but if the validated email later flows into a shell command, log line, or SQL query without further sanitization, the null byte can do damage. Defense in depth would catch it here.

**Fix:** This is solved automatically if M3 is fixed via option 1 (explicit ASCII-printable char class). Otherwise, add an explicit guard: `[^\x00-\x1F\s@]+` instead of `[^\s@]+`.

**L2. reason: 'invalid\_format' is monolithic**

**Where:** Result contract.

A consumer who wants to tell the user *why* their email is malformed gets only "invalid\_format" for six different failure modes. This is a UX limitation, not a correctness bug.

**Fix:** Split into granular reasons (`missing_at`, `missing_domain_dot`, `empty_local_part`, `consecutive_dots`, `whitespace_in_local`, `trailing_dot_in_local`). Adds maybe 15 lines of code and 6 enum values. Skip unless callers actually need it.

**L3. No tests for boundary inputs at the @ itself**

**Where:** Test suite.

Inputs like 'foo@', '@', '@.com', 'foo@.com' aren't explicitly tested. They're *implicitly* covered by `@nodomain.com` and `missing@tld`, but explicit tests would harden against regex edits that loosen the boundary.

**Fix:** Add 2-3 boundary tests in the invalid block.

**L4. Trailing dot in domain (foo@bar.com.) silently passes**

**Where:** Regex behavior I called out as a deliberate permissive choice in the file 2 review.

I documented this as conscious – but there's no test asserting *either* direction. So if a future tightening rejects it, no test will fail; if a tightening accidentally rejects it, no test will fail either. The behavior is undefended.

**Fix:** Add a test that documents the current behavior: `expect(validateEmail('foo@bar.com.')) → valid: true`. If you decide to reject trailing-dot FQDNs later, flip the assertion at that point. This is "lock the contract, whichever way."

```
---
INFO / non-issues (verified clean)

- Catastrophic ReDoS: None. All quantifiers operate on character classes with literal separators between them; backtracking is linear, not exponential.
- CRLF injection (foo@example.com\nBcc: evil@x.com): Rejected. Without the m flag, $ anchors to end-of-string, so embedded newlines fail the match.
- Information leak via reason: None. Reasons are a closed enum; the input is never echoed back.
- JSON load failure: require('./disposable-domains.json') throws at module load if the file is missing/corrupt. That's acceptable – fail-fast at deploy time, not at runtime per request.
- Naming / readability: validateEmail, extractDomain, isDisposableDomain, EMAIL_REGEX, DISPOSABLE_SET all read straight. No issues.
- Test 22 (Object.keys exact match): Catches both missing and extra string-keyed properties. Won't catch symbol keys, but the module never uses symbols, so this is theoretical only.
```

```
---
Recommended next move

If you want to act on this review, I'd prioritize: M2 first (cheap – just tests, locks down current behavior), then M3 option 1 (tightens the regex; simultaneously closes L1), then M1 (length cap). M3 + M1 will require updating a few comments and possibly adding one enum value.
```

Want me to apply any of these? I'll wait for your call before touching code.

\*Worked for 2m 0s

2. Decide which findings are worth fixing. Reply with specific fixes: "Apply fixes for issues 1, 2, and 4."

```
> Apply fixes for issues 1, 2, and 4
```

3. Approve the file modifications one at a time.

```

> Apply fixes for issues 1, 2, and 4

• Reading the current files before editing.

  Read 2 files (ctrl+o to expand)

• Applying three fixes now: M1 (length cap), M2 (tests for documented guards), L1 (control-char hardening).

  Fix 1 – src/validateEmail.js: tighten regex against control chars + update comment.

• Update(src\validateEmail.js)

Opened changes in Visual Studio Code ⓘ

Save file to continue...

Do you want to make this edit to validateEmail.js?
> 1. Yes ←
  2. Yes, allow all edits during this session (shift+tab)
  3. No
  
```

4. After you are done with approving all the changes you will get this output:

```

• All 28 tests pass.

Summary of changes

# | Issue | File | Change
---|---|---|---
M1 | DoS via large inputs | src/validateEmail.js | Added input.length > 320 guard before trim() / regex; rejects as invalid_format to keep the public enum unchanged
M2 | Documented guards untested | tests/validateEmail.test.js | Added 2 tests: consecutive-dots and leading-dot in local part – locks down the (?!.*\.\.) and (?!\.) regex guards
L1 | Control chars slip through | src/validateEmail.js | Tightened the three [^\s@]+ char classes to [^\x00-\x1F\x7F\s@]+ (excludes C0 controls + DEL)
– | New code paths need tests | tests/validateEmail.test.js | Added 2 more tests for the new behaviors: length cap + null byte rejection

Test count: 24 → 28. All passing.

What's left from the review (deliberately not done)

- M3 (Unicode/IDN): Still undefined behavior. 用户@example.com passes the regex but won't match a Punycode blacklist entry. Worth picking a stance before this ships to real users.
- L2 (monolithic invalid_format): Length-cap rejection now shares the same reason as a malformed email. Callers can't tell "you typed 1MB of garbage" apart from "your email is missing an @". UX-only concern.
- L3 (boundary input tests): 'foo@', '@', '@.com' still only implicitly covered.
- L4 (trailing-dot-in-domain test): foo@bar.com. still passes silently with no test asserting either direction.

Want me to handle any of these next, or call it done here?
  
```

5. Type “Call it done here” into your terminal and hit Enter.

```
> call it done here
```

- Done. Final state: 3 files, 28 passing tests, the validator handles strings/non-strings, length-caps oversized input, rejects malformed and control-char-laced addresses, and flags disposable providers including subdomains.

```
> █
```

K21Academy

## 11 PATTERN 5 - VERIFICATION

No task is complete until verification passes. The gold standard is an executable test suite.

### 11.1 Running Tests and Interpreting Results

Verification prompt:

Now write Jest tests that verify the implementation works. Cover:

1. Happy path — a valid normal email returns { valid: true, isDisposable: false }.
2. Disposable detection — a mailinator email returns { valid: true, isDisposable: true }.
3. Invalid format — 'not-an-email' returns { valid: false, reason: 'invalid format' }.
4. Empty input — "" returns { valid: false, reason: 'empty' }.
5. Null input — null returns { valid: false, reason: 'empty' }.

After writing the tests, run them with `npm test` and show me the output.

If any test fails, do not modify the tests — fix the implementation.

### 11.2 Hands-On: Write and Run Verification

#### Tests

1. Paste the verification prompt given above.

```
> Now write Jest tests that verify the implementation works. Cover:
1. Happy path - a valid normal email returns { valid: true, isDisposable: false }.
2. Disposable detection - a mailinator email returns { valid: true, isDisposable: true }.
3. Invalid format - 'not-an-email' returns { valid: false, reason: 'invalid format' }.
4. Empty input - '' returns { valid: false, reason: 'empty' }.
5. Null input - null returns { valid: false, reason: 'empty' }.

After writing the tests, run them with `npm test` and show me the output.
If any test fails, do not modify the tests - fix the implementation.
```

2. Now Claude stopped and pointed out a conflict. The verification prompt from the lab guide was a bit generic (asking for things like `reason: 'empty'`), but Claude had already built a *better, more specific* system during implementation (using `empty_input` and `not_a_string`).

- Pausing before I do anything – this prompt contradicts the contract we just locked down, and I want to flag it before either writing or fixing anything.

**The conflict**

Three of the five tests you described use different reason strings (and different behavior for `null`) than what's currently implemented and tested:

Your new test	Current contract
<code>reason: 'invalid format'</code> (space)	<code>reason: 'invalid_format'</code> (underscore)
<code>'' → reason: 'empty'</code>	<code>'' → reason: 'empty_input'</code>
<code>null → reason: 'empty'</code>	<code>null → reason: 'not_a_string'</code>

The current contract is what we explicitly agreed on in the plan you approved (§6, "reason taxonomy") and what's exported by `src/validateEmail.js` right now. The 28 existing tests in `tests/validateEmail.test.js` all assert against it.

If I follow your instruction literally – *write the new tests, don't modify them, fix the implementation when they fail* – I will:

1. Break all 28 existing passing tests.
2. Switch from `snake_case` to a mix of `'invalid format'` (with space) and `'empty'` (one word). Inconsistent style, and a space inside a `reason` value is unusual.
3. Lose the deliberate `not_a_string` vs. `empty_input` distinction. Today, `null` (programmer bug – passed wrong type) and `''` (user submitted empty form) produce different reasons. Your spec collapses both to `'empty'`.

**Three ways to resolve this**

1. **You meant to redesign the contract.** Confirm and I'll: delete `tests/validateEmail.test.js`, write the 5 new tests as the canonical suite, and rewrite `validateEmail.js` to match the new reason strings. I'd flag #2 and #3 above as design downgrades but defer to your call.
2. **You meant the spirit, not the literal strings.** I'll write the 5 tests against the *existing* contract (`'invalid_format'`, `'empty_input'`, `null → 'not_a_string'`) – same coverage, no implementation changes, all tests pass. You probably noticed our existing 28-test suite already covers all five of these cases and more.
3. **You're using a generic test-writing prompt and didn't notice the conflict.** Same outcome as option 2.

Which one is it?

\* Brewed for 37s

※ **recap:** We built and reviewed a JavaScript email validator (28 passing tests); current task is writing 5 new tests, but your spec uses different reason strings than the existing contract. Next: tell me which of the three resolution options to take.

> █

3. Copy and paste this exact text into your terminal and hit Enter:

Option 2. You meant the spirit, not the literal strings. Keep the existing contract and write the tests to match it.

4. Now, after the prompt gets executed completely. Run “/exit” and exit from the claude code.
5. Now Run “npm test” and hit Enter.
6. You will see a green output showing all your tests passing:

```
PS E:\1. K21 ACADEMY\CLAUDE AI\Claude AI Lab Projects\k21-lab-3-1-patterns> npm test

> k21-lab-3-1-patterns@1.0.0 test
> jest

PASS tests/validateEmail.test.js
  input guards
    ✓ rejects undefined as not_a_string (3 ms)
    ✓ rejects null as not_a_string (1 ms)
    ✓ rejects a number as not_a_string (1 ms)
    ✓ rejects an object as not_a_string (1 ms)
    ✓ rejects empty string as empty_input (1 ms)
    ✓ rejects whitespace-only string as empty_input (1 ms)
  format validation – invalid
    ✓ rejects plainstring (no @) (1 ms)
    ✓ rejects missing@tld (no dot in domain) (1 ms)
    ✓ rejects @nodomain.com (empty local part) (1 ms)
    ✓ rejects "spaces in@local.com" (1 ms)
    ✓ rejects double@@at.com
    ✓ rejects trailing dot in local part (1 ms)
    ✓ rejects consecutive dots in local part
    ✓ rejects leading dot in local part
    ✓ rejects inputs over the length cap (DoS guard)
    ✓ rejects null byte in local part
  format validation – valid, non-disposable
    ✓ accepts alice@example.com (1 ms)
    ✓ accepts plus-addressing (alice+tag@example.com) (1 ms)
    ✓ accepts dotted local + multi-label domain
    ✓ trims surrounding whitespace before validating
  disposable detection
    ✓ flags mailinator.com as disposable
    ✓ flags 10minutemail.com as disposable
    ✓ disposable detection is case-insensitive (1 ms)
    ✓ subdomain of a disposable provider is disposable
    ✓ gmail.com is NOT disposable (sanity check against false positives)

  contract invariants
    ✓ every result has exactly the keys valid, reason, isDisposable (1 ms)
    ✓ reason is always one of the four canonical enum values (2 ms)
    ✓ isDisposable is always false when valid is false (1 ms)

Test Suites: 1 passed, 1 total
Tests:       28 passed, 28 total
Snapshots:   0 total
Time:        0.451 s, estimated 1 s
Ran all test suites.
PS E:\1. K21 ACADEMY\CLAUDE AI\Claude AI Lab Projects\k21-lab-3-1-patterns>
```

---

## 12 PUTTING IT ALL TOGETHER - END-TO-END WORKFLOW

You just applied all five patterns. In production, use this exact sequence for every non-trivial task:

- **Step 1 - Decompose:** prompt Claude to break the request into sub-tasks.
- **Step 2 - Plan:** prompt Claude to convert sub-tasks into a file-level plan.
- **Step 3 - Implement:** approve the plan, then approve each file one at a time.
- **Step 4 - Review:** prompt Claude for self-review + peer-review.
- **Step 5 - Verify:** prompt Claude to write and run tests, read the output yourself.

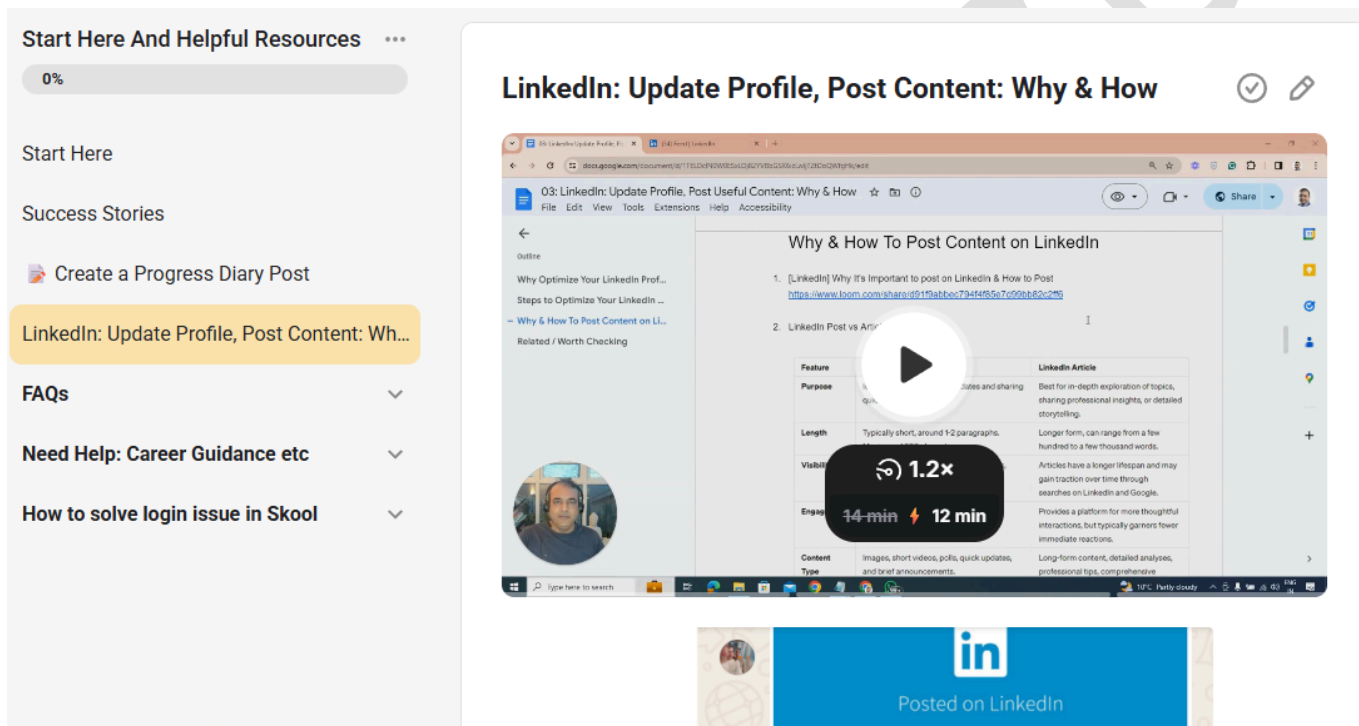
K21Academy

### 13 SHARE YOUR LEARNINGS

In this section, you are going to share whatever you learned during this lab on LinkedIn and the Community

Please watch the video below to understand why it's important to post & how to post:



<https://www.skool.com/k21academy/classroom/8f3838d7?md=d976685318014aafbc658637c00bfc2e>



The screenshot shows a Skool course interface. On the left is a sidebar with a progress bar at 0% and a list of resources including 'Start Here', 'Success Stories', 'Create a Progress Diary Post', 'LinkedIn: Update Profile, Post Content: Wh...', 'FAQs', 'Need Help: Career Guidance etc', and 'How to solve login issue in Skool'. The main content area displays a video player titled 'LinkedIn: Update Profile, Post Content: Why & How'. The video player shows a play button, a 1.2x speed control, and a duration of 14 minutes and 12 seconds. Below the video player is a blue banner with the LinkedIn logo and the text 'Posted on LinkedIn'.

👤 As you know, building a strong professional profile is essential in today's job market. One way to showcase your skills and knowledge is by sharing your labs and projects on LinkedIn.

Your LinkedIn profile is a powerful tool in your job search. Sharing your labs and projects is just one way to demonstrate your expertise and stand out to potential employers. If you don't have LinkedIn, we strongly recommend you create one for yourself.

  By doing so, you can demonstrate to potential employers or connections that you have hands-on experience in your field and are actively engaged in learning and growing your skills.

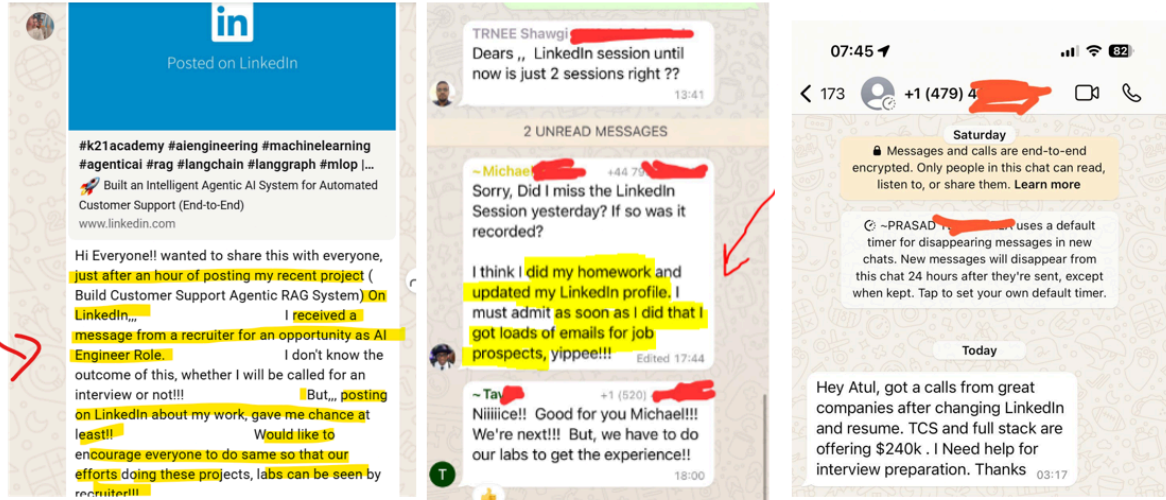
---

## 13.1 Share Hands-On Lab on LinkedIn to Build Authority & Job Opportunities

This will attract recruiters and potential employers to your profile and increase your Authority & LinkedIn reach.

1. **Do remember to tag so we can share too and we'll ask other members to share too (this will increase your post reach)**
  - a. K21Academy ( <https://www.linkedin.com/company/k21academy> ) &
  - b. Atul Kumar ( <https://www.linkedin.com/in/atulk21academy/> ), as we will further circulate in our network too to increase your reach.
  
2. Here is what you can expect
  - a. Jessy got a message from the Recruiter for an AI Engineer Role
  - b. Michael got loads of emails for Job prospects
  - c. Prasad got calls from great companies that are offering \$240K Salary

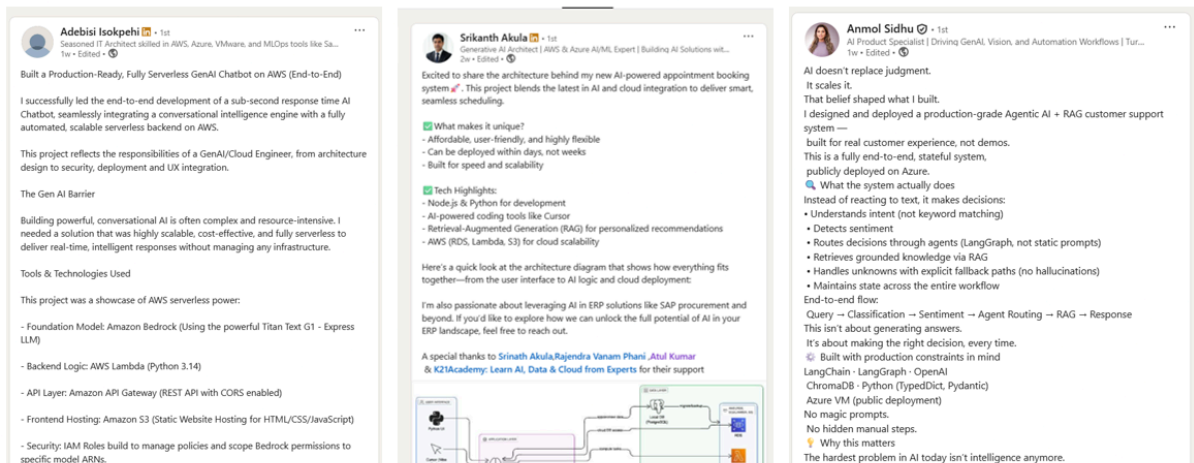
## Results You can Expect



### 3. Example Links on How to post

#### a. Example Posts

## Post on LinkedIn for Interview Calls



#### b. Example Video & Post

Copyright© K21Academy | All Rights Reserved

Sharing, Reselling, or duplication of this content is strictly prohibited without K21Academy's written permission

<https://www.skool.com/k21academy/classroom/0c24b6af>

**LinkedIn for Visibility & Authority** ...

40%

LinkedIn: Update Profile, Post Content: Wh...

FAQ: If I Publish, What Current Employer T...

Why it's Imp. to Prepare & Share Not... ✓

**Example Post: What & How to Post ...** ✓

Useful ChatGPT Prompts ✓

How to create LinkedIn Banner from Canv...

**LinkedIn Profile Review** ^

Neha: LinkedIn Feedback to Updat... ✓

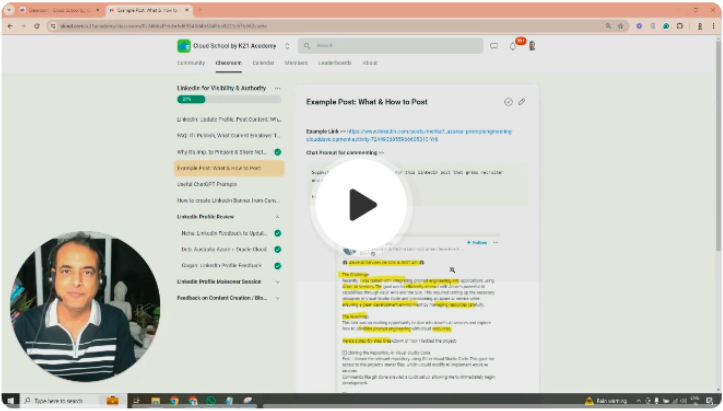
Deb: Australia Azure + Oracle Cloud ✓

Gagan: LinkedIn Profile Feedback ✓

**LinkedIn Profile Makeover Session** v

**Feedback on Content Creation / Blo...** v

### Example Post: What & How to Post & Commenting



**Example Link** >> [https://www.linkedin.com/posts/mehta7\\_azureai-promptengineering-clouddevelopment-activity-7244936855966605313-Yrti](https://www.linkedin.com/posts/mehta7_azureai-promptengineering-clouddevelopment-activity-7244936855966605313-Yrti)

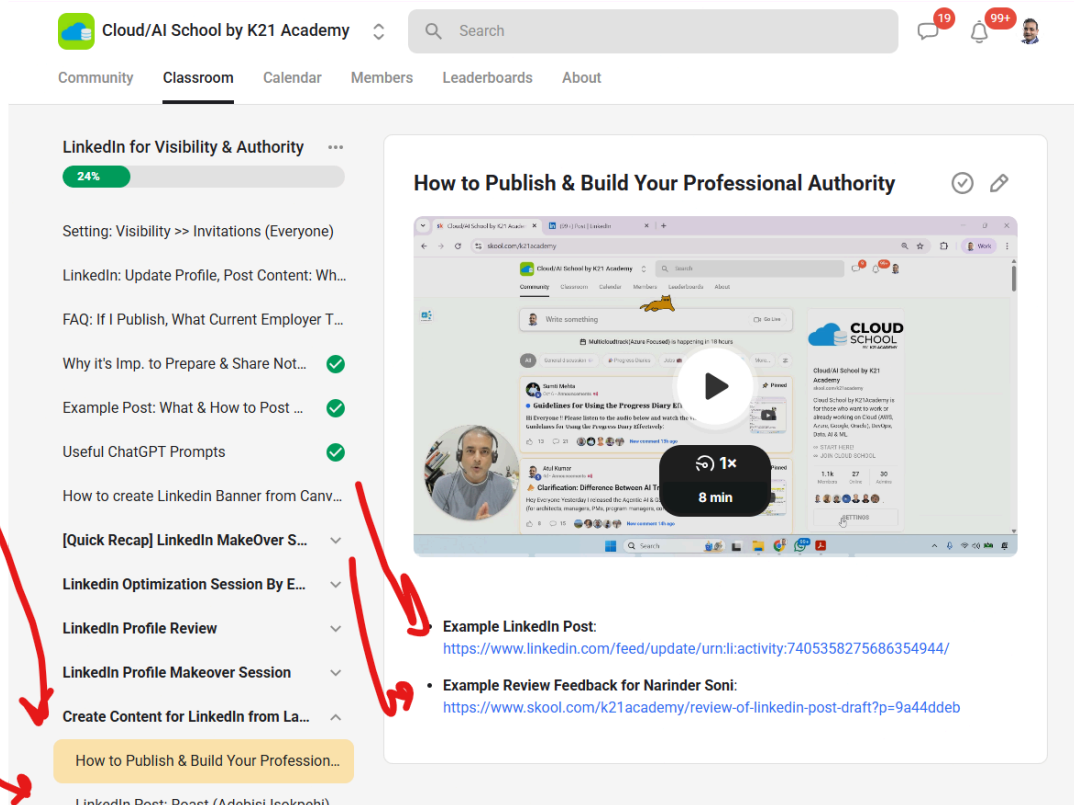
**Chat Prompt for commenting** >>

Suggest useful, engaging comment for this linkedIn post that grabs recruiter and potetnial employers attention

[https://www.linkedin.com/\\*\\*\\*\\*\\*](https://www.linkedin.com/*****)

c. Check few more linkedIn post examples (do's and don'ts)

- <https://www.skool.com/k21academy/classroom/0c24b6af?md=edfe6092fbb84f6bba03ceb439d1e00a>



Cloud/AI School by K21 Academy

Community Classroom Calendar Members Leaderboards About

LinkedIn for Visibility & Authority 24%

Setting: Visibility >> Invitations (Everyone)

LinkedIn: Update Profile, Post Content: Wh...

FAQ: If I Publish, What Current Employer T...

Why It's Imp. to Prepare & Share Not... ✓

Example Post: What & How to Post ... ✓

Useful ChatGPT Prompts ✓

How to create LinkedIn Banner from Carv...

[Quick Recap] LinkedIn MakeOver S... ▾

LinkedIn Optimization Session By E... ▾

LinkedIn Profile Review ▾

LinkedIn Profile Makeover Session ▾

Create Content for LinkedIn from La... ▾

How to Publish & Build Your Profession...

LinkedIn Post: Roast (Adehisi Isoknehi)

How to Publish & Build Your Professional Authority

Example LinkedIn Post:  
<https://www.linkedin.com/feed/update/urn:li:activity:7405358275686354944/>

Example Review Feedback for Narinder Soni:  
<https://www.skool.com/k21academy/review-of-linkedin-post-draft?p=9a44ddeb>

## 13.2 Share the win on the Community via

### Progress Diary

Share screenshots in Cloud School Community under your **'Progress Diary'** ( If you have not created a Progress Diary yet, then check at

<https://k21academy.com/progressDiary>

This will boost your confidence in the progressive path you are following, and encourage/inspire others to perform these hands-on labs.

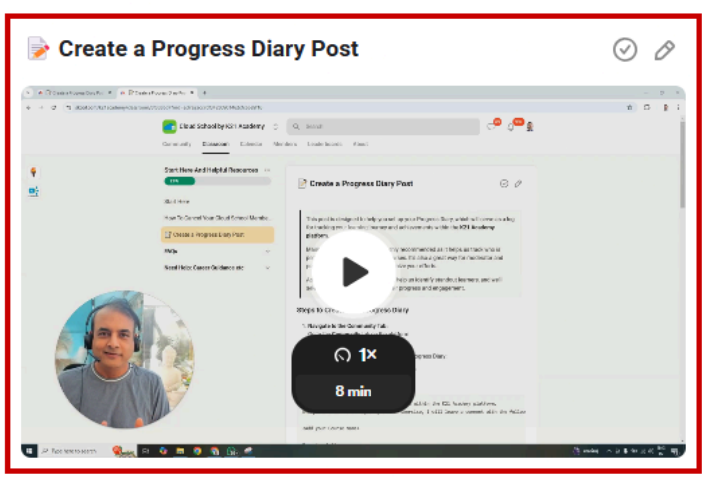
**Start Here And Helpful Resources** ...  
0%

Start Here  
How To Cancel Your Cloud School Membe...

**Create a Progress Diary Post**

**FAQs**

- How to Check Schedule & Live Session ...
- 1. Is there a Mobile App for K21 Acade...
- 2. Progress Tracking for Courses
- 3a. Adjust Video Speed 1.5x or 2.x
- 3b. Screen Quality Adjustment
- 4. How to Create Ubuntu Machine on A...
- 5. How long will we have access to Sko...



This post is designed to help you set up your Progress Diary, which will serve as a log for tracking your learning journey and achievements within the **K21 Academy platform**.

K21Academy

## 14 DELETE AND CLEAN-UP

No cloud resources were provisioned in this lab. Cleanup is optional.

- To close the session, type `/exit` in Claude Code.
- If you want to remove the project entirely, run: `cd ..` then `rm -rf k21-lab-3-1-patterns`

**(Recommendation: Keep the project for reference in upcoming modules).**

K21Academy

## 15 TROUBLESHOOTING

In this section, we outline some common issues you may encounter during the process, along with explanations of why these problems may occur. This will help you quickly identify and resolve any challenges you face.

**Note:** *If you hit any issue in performing this lab, then add your issue in the thread below this lab (linked to this thread in the community)*



### Lab: Task Decomposition, Planning, Implementation, Review & Verification Patterns – Share Your Progress, Issues & Feedback!

Hi Everyone!

If you've recently completed the [Task Decomposition, Planning, Implementation, Review & Verification Patterns Lab](#) – great work! 🍀

Please drop a quick update here to let us know how it went.

💬 **For example, you can post:**

✅ **Success!** "I successfully guided Claude through the Decomposition phase, and it caught edge cases I wouldn't have thought of!" "The Verification step using Jest tests worked perfectly, and my email validator passed all 5 test cases!" "I used the Self-Review prompt, and Claude actually found and fixed a potential Unicode bug in its own code!"



---

## 16 SUMMARY

In this lab, we have covered why patterns matter in agentic development to prevent silent bugs and make review possible. We explored the five core patterns: Decomposition, Planning, Implementation, Review, and Verification. We learned how Claude Code's built-in features (plan mode, approval gates, diffs, test execution) support each pattern.

Learners gained hands-on experience decomposing an email validator feature, generating and approving plans, implementing code file by file, conducting peer reviews, and executing Jest test suites. With Lab 3.1 complete, you understand the foundational workflow for all future agentic development. Next up — Lab 3.2: Building Production Workflows Using Subagents and Pipelines