56 · Machine Coding Rounds

Flutter Practice Notes

Contents

56 · Machine Coding Rounds

Introduction

This module covers the **machine-coding round** — a timed (typically 60–120 min) interview where you **build a small working Flutter feature/app** and are judged on **functionality + code quality + structure**, not just "does it run." It walks the **fundamentals** (format, evaluation criteria, mindset), the **approach & time management** (scope → plan → build incrementally → polish), **clean architecture under time pressure** (right-sized structure, state-management choice, separation without over-engineering), **common problems + patterns** (todo/timer/search/form/paginated list…), and a **worked-round capstone**. It applies state management (Module 11), MVVM/Clean (Module 43/Module 40), and testing (Module 49) under a clock.

Why this module exists

Machine coding is where strong engineers **win or lose offers**: it's not about clever algorithms but about **shipping a clean, working feature fast** — scoping requirements, picking the right state approach, structuring code that's readable and extensible, handling edge cases, and communicating throughout — all under time pressure. Common failures are **over-engineering** (no time to finish), **under-structuring** (a god-widget mess), **poor scoping** (building the wrong thing), and **not finishing**. This module gives the approach, the right-sized architecture, and the practiced patterns to consistently produce a solid submission.

Module map

#	File	Topic	Level
1	01_machine_coding_fundamentals.md	Format, evaluation criteria, mindset	🔵
2	02_approach_and_time_management.md	Scope → plan → incremental build → polish; time-boxing	🔴
3	03_clean_architecture_under_pressure.md	Right-sized structure, state choice, separation w/o over-engineering	🔴
4	04_common_problems_and_patterns.md	Common problems (todo/timer/search/form/list) + reusable patterns	🟡
5	05_machine_coding_integration.md	Capstone: a worked machine-coding round	🔴

> **Cross-references:** Interview prep (round context): Module 55. State management: Module 11. MVVM/Clean (right-sized): Module 43/Module 40. Testing: Module 49. Networking (if API-backed): Module 16. Error handling: Module 38.

Prerequisites

55 Flutter Interview Preparation, 11 State Management, 43 MVVM, 40 Clean Architecture, 49 Testing.

What you'll be able to do after this module

- Explain the machine-coding format + how it's evaluated, with the right mindset.
- Run a reliable approach: scope, plan, build incrementally, and polish within the time box.
- Apply right-sized architecture (state choice + separation) without over-engineering.
- Recognize + solve common machine-coding problems with reusable patterns.
- Execute a full worked round that finishes a clean, working, extensible feature.

Capstone

Worked round: A full simulated machine-coding round for a common prompt (e.g., a searchable/paginated list or a todo app) — clarify + scope, plan the increments, build a working feature with right-sized architecture (state management + clean separation), handle edge cases (loading/empty/error), add a light test or two, and communicate throughout — finished within a realistic time box, with a reflection on the evaluation criteria.

Summary

Machine coding tests shipping a clean, working feature fast: scope tightly, pick the right state approach, structure code right-sized (not over/under-engineered), handle edge cases, finish, and communicate. Practicing the approach + common patterns turns time pressure into a consistent, offer-winning submission.

Machine Coding Fundamentals

> A machine-coding round asks you to **build a small, working Flutter feature/app in a time box** (~60–120 min), evaluated on **five axes**: **functionality** (does it work + handle edge cases?), **code quality** (readable, idiomatic, well-named), **structure/architecture** (right-sized separation, sensible state management), **completeness** (finished + polished, not a half-feature), and **communication** (thinking aloud, clarifying, explaining choices). The winning mindset: **a finished, clean, well-structured small feature beats an ambitious unfinished mess** — scope tightly, build incrementally, keep it right-sized, and finish. It's an engineering-under-time test, not an algorithms puzzle.

Introduction

This file establishes what machine coding is, the evaluation criteria, and the mindset — the orientation before the approach/architecture/patterns files. It reframes the round from "code fast" to "ship a clean, working feature deliberately."

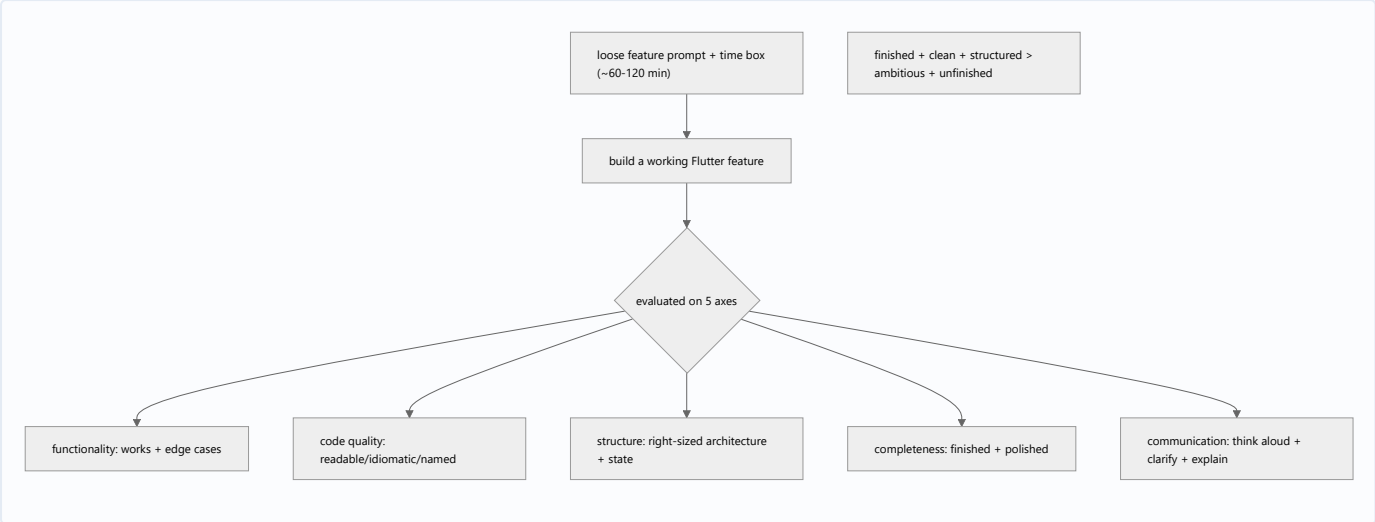
Why this concept exists

Machine coding predicts real on-the-job ability better than algorithm puzzles: can you take a loose spec and produce **working, maintainable, extensible** code under normal time constraints? Knowing exactly **how it's scored** (not just "make it run") lets you allocate effort correctly — because clean structure + a finished feature + good communication routinely beat a clever-but-broken or messy submission.

Real-world analogy

It's a **timed cooking round judged like a restaurant service**, not a speed-eating contest: the judges taste whether the dish **works** (functionality), is **well-prepared/presented** (code quality), used a **sensible kitchen workflow** (structure), is **actually finished + plated** (completeness), and whether you **communicated your plan** as you cooked. An ambitious dish left half-raw scores worse than a simple dish executed cleanly and finished on time.

Internal Working



- **The format**: a **loose feature spec** ("build a todo app", "a searchable paginated list", "a stopwatch") + a **time box** (~1–2h), often live (screen-shared, thinking aloud) or take-home. You produce **running Flutter code**; sometimes an API is provided, sometimes mocked/local. You choose the state approach + structure.
- **Evaluation criteria (know all five — effort follows them)**:
  1. **Functionality**: does it **work** and handle **edge cases** (loading/empty/error, invalid input, boundaries)? The baseline.
  2. **Code quality**: **readable, idiomatic Dart/Flutter**, good names, small widgets/methods, no dead/duplicate code, consistent style.
  3. **Structure/architecture**: **right-sized separation** (UI vs logic vs data), a sensible **state-management** choice, extensibility — **not** a god-widget, **not** over-engineered.
  4. **Completeness**: a **finished, polished** feature (works end-to-end, reasonable UX) — **finishing matters more than scope**.
  5. **Communication**: **clarify requirements**, **think aloud**, explain decisions/trade-offs, respond to hints — assessed throughout (esp. live rounds).
- **The mindset (the key)**: **a finished, clean, well-structured small feature beats an ambitious unfinished mess.** Optimize for **shipping something solid**: scope tight, prioritize the core happy path + edge cases, keep architecture **right-sized**, and **finish + polish** — leaving nothing broken.
- **What it's NOT**: not an algorithms/DSA puzzle (that's the coding round — Module 55); cleverness isn't rewarded — **engineering judgment under time** is.
- **Common failure modes** (avoid — detailed in later files):
  - **Over-engineering**: full Clean Architecture + 5 layers for a todo app → no time to finish.
  - **Under-structuring**: everything in one giant `build` / `setState` god-widget → unreadable, unextensible.

    - **Poor scoping**: building gold-plated extras before the core works, or misreading the spec.
    - **Not finishing**: an impressive-but-broken half-feature.
    - **Silent coding**: no clarification/communication.
- **The winning shape**: clarify → scope → plan increments → build core (right-sized) → handle edge cases → polish → (light test if time) → communicate throughout (02_approach_and_time_management.md).
- **Levels**: junior → a working feature with basic structure; mid → clean + handles edge cases + sensible state; senior → clean, extensible, well-architected, tested, with clear trade-off reasoning (Module 55).

Memory Representation

Not code — a **scoring model**: the five axes you're evaluated on, and a mental priority (functionality + completeness + structure + quality + communication). The submission is a small, finished, clean Flutter feature.

Compiler / Runtime / Engine / VM Behavior

Not applicable beyond running your Flutter code normally — machine coding is an engineering-judgment-under-time test, not a runtime/internals exam.

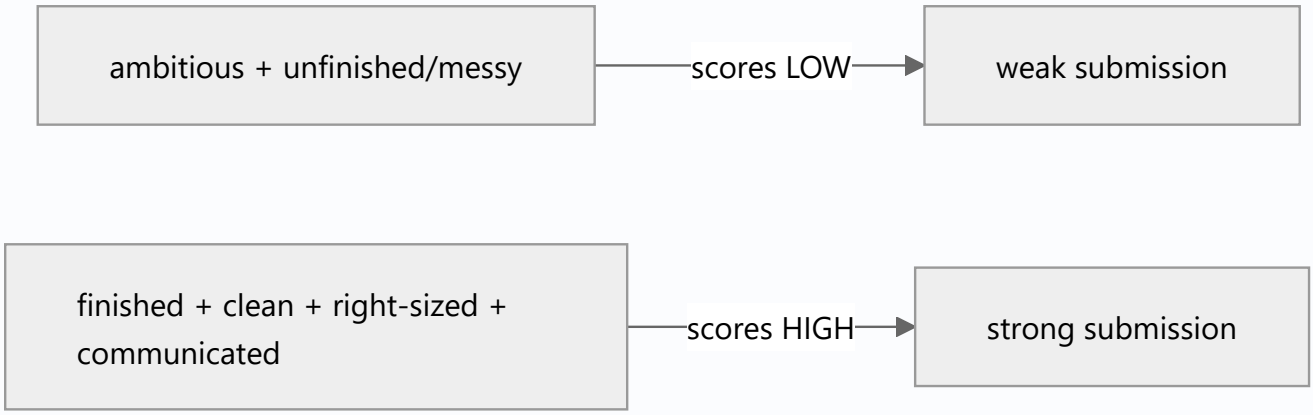
Examples

```
Evaluation axes (allocate effort here):
  functionality  -> works + edge cases (loading/empty/error/invalid)      [baseline]
  completeness   -> finished + polished end-to-end                        [finishing > scope]
  structure      -> right-sized separation + sensible state mgmt          [not god-widget, not over-eng]
  code quality   -> readable/idiomatic/named/small/no-duplication
  communication  -> clarify + think aloud + explain trade-offs            [throughout]

Mindset: a FINISHED, clean, small feature > an AMBITIOUS, unfinished mess.

Failure modes to avoid:
  over-engineering (5 layers for a todo) | under-structuring (god-widget) |
  poor scoping (gold-plating before core works) | not finishing | silent coding
```

Diagrams



Common Mistakes

Mistake	Why it scores low	Fix
Over-engineering (full Clean Arch for a todo)	No time to finish	Right-size architecture to the task
God-widget (all in one `build` / `setState`)	Unreadable/unextensible	Separate UI vs logic vs data
Not finishing	Incomplete = low functionality/completeness	Scope tight; finish core first
Gold-plating before core works	Wrong priorities	Core happy path + edge cases first
Ignoring edge cases (loading/empty/error)	Fails "functionality"	Handle them explicitly
Silent coding	Misses communication axis	Clarify + think aloud + explain
Treating it as an algo puzzle	Wrong lens	Engineering judgment under time
Messy/unnamed code	Fails "code quality"	Readable, idiomatic, well-named

Best Practices

- Optimize for the **five axes** (functionality, code quality, structure, completeness, communication) — allocate effort where the score is; **finishing a clean small feature beats an unfinished ambitious one**.
- Keep architecture **right-sized** (separation without over-engineering; sensible state choice); **handle edge cases** (loading/empty/error/invalid); write **readable, idiomatic** code.
- **Clarify + think aloud + explain trade-offs** throughout (communication is scored); **scope tightly** and **finish + polish** the core before extras.

- Treat it as **engineering-under-time** (not an algorithms puzzle); calibrate depth to your **target level**.

Performance

Not runtime perf — the "performance" is your **submission score**: a finished, clean, well-structured, communicated feature. Time is the constraint, so effort must track the scoring axes (finish + structure + edge cases + communication), not cleverness.

Advantages / Disadvantages

- **+** (As a test) predicts real engineering ability (build clean working features under time); rewards judgment/scoping/communication; practicable.
- **−** Time pressure + loose specs; easy to over/under-engineer or not finish; communication + edge cases often neglected under stress.

Interview Questions

1. 🟢 **What is a machine-coding round, and what does it test?** — Building a small working Flutter feature in a time box, evaluated on functionality, code quality, structure/architecture, completeness, and communication — engineering judgment under time, not algorithms.
2. 🟢 **What's the core mindset?** — A finished, clean, well-structured small feature beats an ambitious unfinished mess — scope tight, keep it right-sized, and finish.
3. 🟡 **What are the five evaluation axes?** — Functionality (works + edge cases), code quality (readable/idiomatic), structure (right-sized separation + state), completeness (finished/polished), communication (clarify/think aloud/explain).
4. 🟡 **What are the common failure modes?** — Over-engineering (no time to finish), under-structuring (god-widget), poor scoping (gold-plating), not finishing, and silent coding.
5. 🟡 **How does machine coding differ from the DSA/coding round?** — Machine coding tests building clean working features (structure/state/edge cases/completeness); the coding round tests algorithms/complexity.
6. 🔴 **Why does finishing matter more than scope?** — An unfinished/broken feature fails functionality + completeness (the biggest axes); a smaller finished feature demonstrates working, clean, complete engineering.
7. 🔴 **How do expectations scale by level?** — Junior: working + basic structure; mid: clean + edge cases + sensible state; senior: clean/extensible/well-architected/tested + trade-off reasoning.

Senior Engineer Tips

- Anchor on the five axes and the "finished beats ambitious" rule; the most common way strong candidates fail is over-engineering or over-scoping and not finishing a clean, working feature.
- Handle loading/empty/error and communicate throughout even in a tiny app; those two (edge cases + communication) are constantly neglected under time pressure and cheaply win points.
- Calibrate scope + architecture depth to the time box and your target level — a right-sized, finished, well-named feature signals exactly the on-the-job judgment they're hiring for.

Architect Perspective

Machine coding is the interview's closest proxy for real work: produce a clean, working, right-sized, finished feature under time while communicating. It rewards the handbook's core judgment — right-sizing architecture, choosing state management, handling edge cases, prioritizing — rather than cleverness. Understanding the five scoring axes and the "finished + clean beats ambitious + broken" mindset is the foundation for the approach, architecture, and patterns that make submissions consistently strong (02_approach_and_time_management.md, 03_clean_architecture_under_pressure.md, Module 55).

Summary

- Machine coding = build a small working Flutter feature in a time box, scored on functionality, code quality, structure, completeness, and communication.
- Mindset: a finished, clean, right-sized small feature beats an ambitious unfinished mess — scope tight, handle edge cases, finish + polish, communicate.
- Avoid over-engineering / under-structuring / poor scoping / not finishing / silent coding; it's engineering-under-time, not an algo puzzle; calibrate to level.

Revision Notes

- Format: loose feature spec + time box (~60-120 min), running Flutter code, you choose state/structure; live (think-aloud) or take-home.
- Five axes: functionality (works + edge cases), code quality (readable/idiomatic/named), structure (right-sized separation + state mgmt), completeness (finished/polished), communication (clarify/think aloud/explain).
- Mindset: finished+clean+right-sized > ambitious+unfinished. Failures: over-engineering, god-widget, poor scoping/gold-plating, not finishing, silent coding. Not a DSA puzzle; level-calibrated (junior working+basic → senior clean/extensible/tested).

Practice Questions

1. What are the five evaluation axes, and how should they guide effort?
2. Why does a finished small feature beat an ambitious unfinished one?
3. What are the common machine-coding failure modes?

Coding Questions

1. Given a prompt, list the five-axis priorities + a scope for the time box.
2. Identify over-engineering vs under-structuring in a sample submission.
3. Enumerate the edge cases (loading/empty/error/invalid) a prompt needs.

Mini Project

Evaluation-criteria calibration (prep): For two common prompts (e.g., a todo app and a searchable list), write for each: the five-axis priorities, a right-sized scope for a ~90-min box, the edge cases to handle, and the over-engineering/under-structuring traps to avoid — plus how the bar shifts for junior vs senior. Acceptance: five axes applied per prompt; right-sized scope (finish-focused); edge cases enumerated; failure-mode traps identified; level calibration; "finished + clean > ambitious + unfinished" mindset evident.

Approach & Time Management

> Machine coding is won by a **disciplined approach under a clock**: **(1) clarify + scope** (5–10 min: confirm requirements, list must-haves vs nice-to-haves, state assumptions), **(2) plan** (sketch data model, state approach, widget breakdown, increments), **(3) build incrementally** (a working vertical slice first, then layer features — always keep it running/compiling), **(4) handle edge cases** (loading/empty/error/invalid), **(5) polish + finish** (reserve time; leave nothing broken), communicating throughout. The golden rules: **build a working thing early and keep it working**, **prioritize core over gold-plating**, **time-box aggressively**, and **always finish with something solid**.

Introduction

This file is the process for executing a machine-coding round within the time box: clarify/scope → plan → incremental build → edge cases → polish, with time-boxing and communication. It operationalizes the "finish clean, right-sized" mindset (01_machine_coding_fundamentals.md).

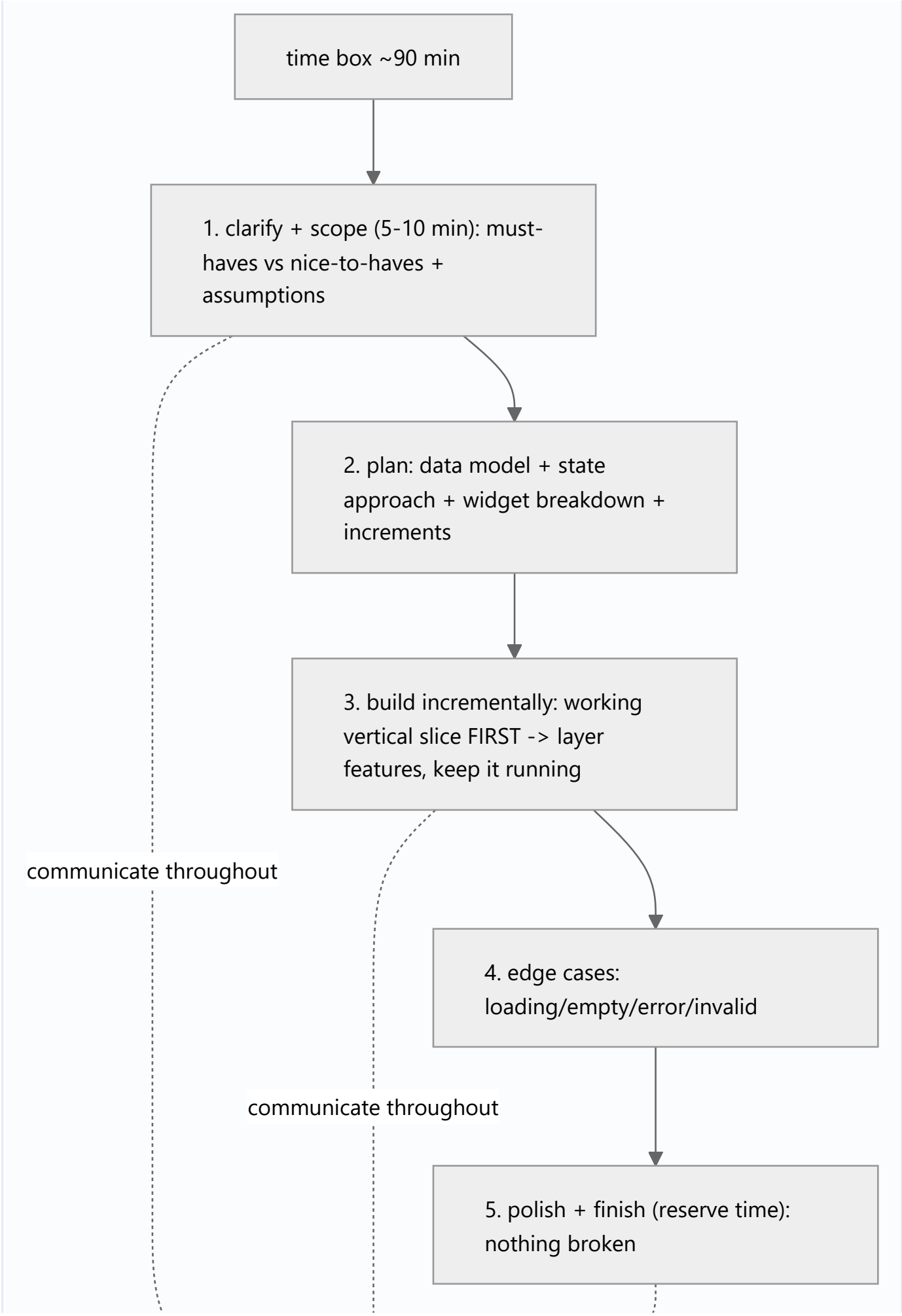
Why this concept exists

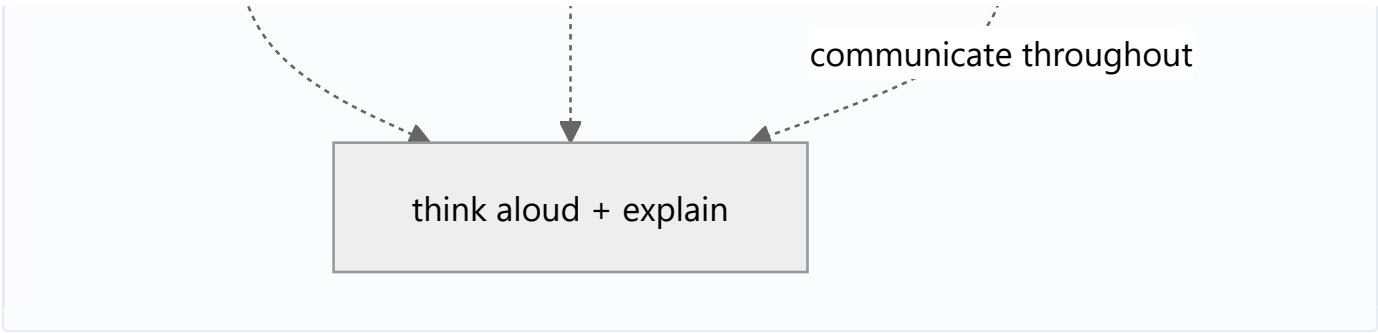
Without a plan, candidates thrash: they mis-scope, start coding blind, over-build one part, and run out of time with a broken app. A repeatable approach + aggressive time-boxing ensures you **always have a working, finishable submission** and spend effort on what's scored — the difference between a polished feature and a half-done mess at the buzzer.

Real-world analogy

It's a **timed kitchen service**: first confirm the order + plan the courses (clarify/scope/plan), get a **simple plate out fast and keep the line moving** (working slice early), then add courses (incremental features), taste for problems (edge cases), and **plate + garnish before time's up** (polish/finish) — narrating to the judges throughout. Chefs who start cooking without a plan, or perfect one dish while others burn, fail the service.

Internal Working


time box ~90 min
1. clarify + scope (5-10 min): must-haves vs nice-to-haves + assumptions
2. plan: data model + state approach + widget breakdown + increments
3. build incrementally: working vertical slice FIRST -> layer features, keep it running
4. edge cases: loading/empty/error/invalid
5. polish + finish (reserve time): nothing broken
communicate throughout
communicate throughout



- **1. Clarify + scope (5–10 min — don't skip)**:
  - **Confirm requirements**: ask about the feature, data source (API/mock/local), constraints, and expected behavior — **don't assume**; misreading the spec is fatal.
  - **List must-haves vs nice-to-haves**: identify the **core** (the thing that must work) and defer extras. **State assumptions** aloud ("I'll assume local data, no persistence").
  - This upfront time **pays back** by preventing building the wrong thing.
- **2. Plan (brief, explicit)**:
  - Sketch the **data model** (entities), the **state approach** (setState/ChangeNotifier/Provider/Bloc — right-sized — 03_clean_architecture_under_pressure.md), the **widget breakdown** (screens/components), and the **increment order** (what to build first→last).
  - A 2–3 min plan prevents thrashing; announce it (communication + a shared roadmap with the interviewer).
- **3. Build incrementally (the core discipline)**:
  - **Get a working vertical slice out fast** — the simplest end-to-end thing (e.g., a static list rendering, then add data, then interactions). **Always keep it compiling/running** so you're never far from a demoable state.
  - **Layer features** one at a time (add → verify → next), core first. Avoid long stretches of broken/non-compiling code.
  - This guarantees you **always have something to show** and can stop at any point with a working submission.
- **4. Handle edge cases**: once the happy path works, add **loading/empty/error** states, **input validation**, and boundary handling (Module 38). These are scored under **functionality** and often neglected.
- **5. Polish + finish (reserve time)**: **budget the last ~10–15%** for polish — remove dead code, fix naming, tidy UI, quick self-test, and ensure **nothing is broken**. A finished, clean submission is the goal; a broken half-feature isn't.
- **Time-boxing (aggressive)**:
  - Roughly: **~10% clarify/plan, ~60% core build, ~15% edge cases, ~15% polish/finish** (adjust to length). **Watch the clock**; if behind, **cut nice-to-haves**, not correctness/finishing.

    - **Don't rabbit-hole**: if stuck on one detail, **stub/simplify it** and move on (note it aloud) — finishing the whole > perfecting one part.

- **Communicate throughout** (scored): narrate your plan, decisions, trade-offs, and what you're cutting; ask when unsure; respond to hints. It shows engineering judgment + collaboration.
- **If time runs short**: **finish a smaller scope cleanly** rather than leaving the full scope broken — always land on a working, polished slice.

Memory Representation

Not code — a **time-boxed plan**: phase budget (clarify/plan/build/edge/polish), a must-have/nice-to-have list, a data-model/state/widget/increment sketch, and a running-slice checkpoint you protect. The submission stays demoable at every step.

Compiler / Runtime Behavior

Keeping the app **compiling/running** at each increment is the discipline — you're never in a "big broken refactor" with no working state. Run it frequently to verify.

Flutter Engine / Dart VM Behavior

Not applicable beyond running your Flutter app normally (hot reload speeds the incremental loop).

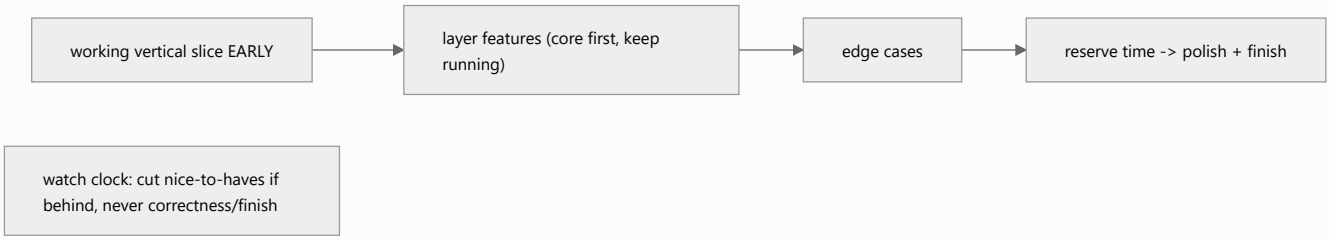
Examples

```
Time box (~90 min) — phase budget:
  0-10   clarify + scope (must-haves vs nice-to-haves, assumptions) + plan (model/state/widgets/increments)
  10-65  build core incrementally (working slice first, layer features, keep it running)
  65-80  edge cases (loading/empty/error/invalid)
  80-90  polish + finish (dead code, naming, UI, self-test — nothing broken)
  throughout: communicate (plan, decisions, cuts)

Increment order example (searchable list):
  1) render a static list  2) load data (mock)  3) show loading/empty/error
  4) add search filter  5) polish UI  (+pagination only if time = nice-to-have)
  -> app RUNS after every step; stop anytime with a working submission
```

```
Rules:
  working thing EARLY + keep it working | core before gold-plating | time-box aggressively |
  don't rabbit-hole (stub + move on) | reserve polish time | ALWAYS finish something solid |
  communicate + state assumptions throughout
```

Diagrams



Common Mistakes

Mistake	Why it fails	Fix
Skipping clarify/scope	Builds the wrong thing	5-10 min clarify + must/nice-to-have + assumptions
Coding with no plan	Thrashing, wasted time	Brief plan (model/state/widgets/increments)
Big broken stretches (not running)	Nothing to show if time ends	Incremental slices, keep it compiling/running
Gold-plating before core works	Wrong priorities, no finish	Core happy path first; extras only if time
Rabbit-holing one detail	Blows the budget	Stub/simplify + move on (note it)
No time reserved for polish	Messy/broken at buzzer	Reserve last ~10-15% for polish/finish
Ignoring edge cases	Fails functionality	Add loading/empty/error/invalid
Silent execution	Misses communication	Narrate plan/decisions/cuts throughout

Best Practices

- **Clarify + scope first** (5–10 min: must-haves vs nice-to-haves + assumptions), then **plan** (data model, state approach, widget breakdown, increment order) — announce both.
- **Build incrementally**: a **working vertical slice early**, then **layer features (core first)** while **keeping it running** — so you're always demoable.
- **Handle edge cases** (loading/empty/error/invalid) after the happy path; **reserve ~10–15% for polish/finish** (nothing broken); **time-box aggressively** and **cut nice-to-haves (not correctness/finishing)** if behind.
- **Don't rabbit-hole** (stub/simplify + move on); **communicate throughout**; if short on time, **finish a smaller scope cleanly**.

Performance

Not runtime perf — the "performance" is finishing a clean feature within the box. The approach maximizes it: upfront scoping avoids wasted work, incremental slices keep you demoable, aggressive time-boxing prevents the buzzer catching you broken. Hot reload keeps the build loop fast.

Advantages / Disadvantages

- **+** Always-demoable submission, correct scoping, finished + polished result, effort on scored axes, calm execution under pressure.
- **−** Requires discipline (resisting gold-plating/rabbit-holes), upfront scoping/planning time, honest time-boxing + cutting scope when behind.

Interview Questions

1. 🟢 **What's the first thing you do in a machine-coding round?** — Clarify requirements + scope (must-haves vs nice-to-haves, state assumptions) — 5–10 min — before coding, to avoid building the wrong thing.
2. 🟢 **Why build a working vertical slice early?** — So the app is always demoable/runnable; you can stop at any point with something working, rather than risk a broken half-feature at the buzzer.
3. 🟡 **How do you time-box a ~90-min round?** — Roughly ~10% clarify/plan, ~60% core build, ~15% edge cases, ~15% polish/finish; watch the clock and cut nice-to-haves (not correctness/finishing) if behind.
4. 🟡 **What do you do when stuck on one detail?** — Stub/simplify it and move on (noting it aloud); finishing the whole feature beats perfecting one part.
5. 🟡 **When do you add edge cases + polish?** — Edge cases (loading/empty/error/invalid) after the happy path works; reserve the last ~10–15% for polish/finish so nothing is broken.
6. 🔴 **What if you're running out of time?** — Finish a smaller scope cleanly (cut nice-to-haves) rather than leaving the full scope broken — always land on a working, polished slice.
7. 🔴 **Why communicate throughout, and what do you communicate?** — It's scored (judgment/collaboration): narrate your plan, decisions, trade-offs, assumptions, and what you're cutting; ask when unsure.

Senior Engineer Tips

- Spend the first 5–10 minutes clarifying + planning and announce your scope/increments; it feels like "not coding" but it's the highest-ROI time — it prevents building the wrong thing and shows judgment.

- Keep the app compiling/running after every small increment and reserve the last chunk for polish; the candidates who fail usually have a broken, half-refactored app when time is called.
- Resist rabbit holes and gold-plating: stub the hard/optional bits, finish the core cleanly, and cut nice-to-haves before correctness — a smaller finished feature always beats an ambitious broken one.

Architect Perspective

The approach is time-boxed engineering discipline: scope tightly, plan briefly, build in always-running increments (core first), handle edge cases, and reserve time to finish + polish — communicating throughout. It's the same prioritization + right-sizing the handbook teaches, compressed under a clock, ensuring you always deliver a clean, working, finishable feature. Combined with right-sized architecture and practiced patterns, it makes machine-coding outcomes consistent rather than luck-of-the-clock (03_clean_architecture_under_pressure.md, 04_common_problems_and_patterns.md, 01_machine_coding_fundamentals.md).

Summary

- Approach: clarify + scope (must/nice-to-have + assumptions) → plan (model/state/widgets/increments) → build incrementally (working slice first, keep running, core first) → edge cases → reserve time to polish + finish.
- Time-box aggressively (~10/60/15/15), don't rabbit-hole (stub + move on), cut nice-to-haves (not correctness/finishing) if behind, and communicate throughout.
- Always finish something solid — a smaller clean feature beats an ambitious broken one.

Revision Notes

- Phases (time-box ~90min): clarify+scope 5-10min (must/nice-to-have + assumptions) → plan (data model/state approach/widget breakdown/increment order) → build incrementally (working vertical slice FIRST, layer core-first, keep compiling/running) → edge cases (loading/empty/error/invalid) → polish+finish (reserve ~10-15%, nothing broken).
- Rules: working thing early + keep working; core before gold-plating; aggressive time-box (~10/60/15/15); don't rabbit-hole (stub+move on); cut nice-to-haves not correctness/finish if behind; ALWAYS finish something solid; communicate (plan/decisions/cuts/assumptions) throughout; hot reload speeds the loop.

Practice Questions

1. What's the phase breakdown + time budget for a ~90-min round?
2. Why keep the app running at every increment?

3. What do you cut when behind, and what do you never cut?

Coding Questions

1. For a given prompt, write the clarify questions + scope + increment plan.
2. Order the build increments so the app runs after each step.
3. Define the phase time budget + what you'd cut if 20 min behind.

Mini Project

Timed approach plan (prep): For a common prompt (e.g., a searchable list), write a full execution plan for a ~90-min box: clarify questions + scope (must vs nice-to-have + assumptions), a brief plan (data model/state approach/widget breakdown), an ordered increment list (app runs after each), an edge-case list, a reserved polish/finish step, and the time budget + what you'd cut if behind. Acceptance: clarify-first + scoped (must/nice-to-have + assumptions); increment order keeps the app running (working slice early, core first); edge cases + reserved polish; time-boxed with a cut-list (never correctness/finishing); communication points noted.

Clean Architecture Under Pressure

> Under a clock you need **just enough structure** — enough to score "well-structured/extensible" without burning time you need to finish. The reliable middle ground: **MVVM-lite** — separate **UI (thin widgets)** from **logic/state (a view model / `ChangeNotifier` or Cubit)** from **data (a simple repository/service)**, pick the **simplest state approach that fits** (`setState` for trivial local UI; `ChangeNotifier` + `Provider` or a Cubit for anything shared/async), keep a **flat, sensible folder structure**, and **stop layering there** — no full Clean Architecture (use cases/DTOs/interfaces) for a 90-minute app. The rule: **separate concerns, right-sized** — not a god-widget, not five layers.

Introduction

This file covers right-sized architecture for machine coding: the MVVM-lite separation, choosing state management under time, folder structure, and where to *stop* layering. It applies MVVM/Clean/state-management (Module 43/Module 40/Module 11) with the volume turned down.

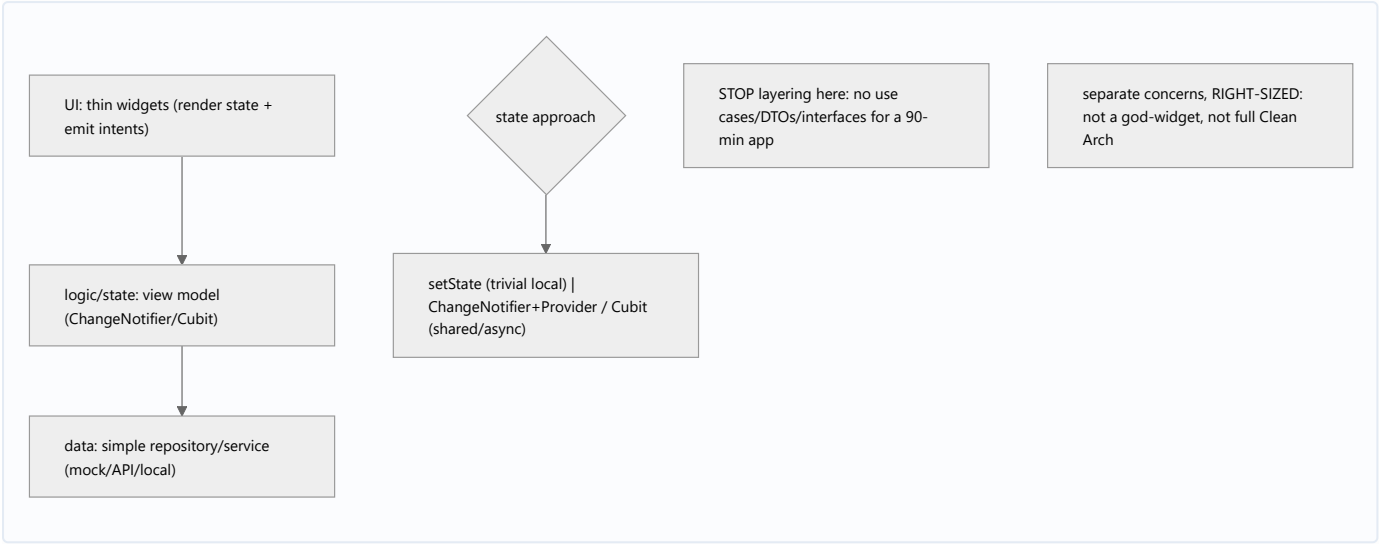
Why this concept exists

The structure axis is scored, but so is finishing — and the two most common failures are **opposite**: a **god-widget** (no separation → unreadable/unextensible → low structure score) and **over-engineering** (full Clean Architecture → no time to finish → low completeness). Right-sizing threads the needle: clear separation of UI/logic/data with the **minimum ceremony**, so you demonstrate architecture *and* finish.

Real-world analogy

It's **organizing a pop-up shop for a weekend**, not building a permanent department store: you still separate the **counter (UI)**, the **register/logic (state)**, and the **stockroom (data)** so it runs cleanly — but you don't install a full corporate org chart, five management layers, and a logistics ERP for a two-day event. Right-sized separation makes it work + look professional without over-building for the timeframe.

Internal Working



- **The right-sized target — MVVM-lite** (Module 43):
  - **UI (thin widgets)**: render state + forward intents (button → `vm.add()`); no business logic, no direct data calls. Break big screens into small widgets (readable + reusable).
  - **Logic/state (view model)**: a `ChangeNotifier`/Cubit holding state + operations, calling the data layer, exposing observable state the UI binds to. **UI-free** (testable).
  - **Data (simple repository/service)**: a plain class fetching/persisting (mock list, `http`/`dio` call, or local store). Return domain-ish models — **skip DTOs/mappers unless the API demands it**.
  - This gives **clear separation (scored)** with **minimal ceremony (finishable)**.
- **Choosing state management under time** (Module 11):
  - `setState`: fine for **trivial, purely-local** UI state (a toggle, a single screen with no shared/async logic). Don't over-reach for a package when `setState` suffices.
  - `ChangeNotifier` + `Provider` (or **Cubit**): the **default** for anything with **shared state, async, or multiple widgets** — clean, low-boilerplate, testable, familiar to interviewers. **Recommended** for most machine-coding prompts.
  - **Bloc/Riverpod**: fine **if you're fluent** and the prompt is complex — but **don't pick an unfamiliar tool under time**; pick what you can move fastest in cleanly. Interviewers care about **clean state handling**, not a specific library.
  - Whatever you pick: **scope rebuilds**, expose **immutable-ish state**, keep the view model UI-free.
- **Folder structure (flat + sensible)**: a small feature-oriented layout — `models/`, `<feature>_view_model.dart` (or a `state/` folder), `<feature>_repository.dart`, `widgets/`, `screens/`. **Don't** create a deep domain/data/presentation × feature matrix for a tiny app — keep it **navigable in seconds**.

- **Where to STOP layering (crucial)**: for a ~90-min app, **skip** full Clean Architecture — **no use cases/interactors, no DTO↔entity mappers, no repository interfaces + DI containers**, unless the prompt is large/senior-level. MVVM-lite (UI/VM/repo) is enough. **Mention** you'd add layers "if this grew" (shows judgment) but **don't build them now**.
- **Right-sizing by prompt/level**: trivial (a counter) → `setState` in a well-organized widget; typical (list+search / todo / form) → **MVVM-lite + ChangeNotifier/Cubit**; large/senior take-home → add a use-case/interface layer + tests if time. **Match structure to scope + level**.
- **Edge cases live in the view model/state** (loading/empty/error as explicit states — Module 38); the UI renders each. This is both structure + functionality points.
- **Testability signal (bonus)**: because the view model is UI-free, a **quick unit test** (or two) of a state transition is a strong signal if time allows — but **finish the feature first** (Module 49).

Memory Representation

Not new structures — a **layer map**: thin widgets ↔ a UI-free view model (state + ops) ↔ a simple repository/service, in a flat feature folder. State lives in the view model (observable); UI holds none. No use-case/DTO/interface layers unless justified.

Compiler Behavior

The view model is plain Dart (compiles UI-free → testable); the UI binds to it. Keeping separation compile-clean (no widget imports in the VM) is the checkable boundary — cheap to maintain.

Runtime Behavior

Intent → VM op → (repo) → state update → UI rebuilds (scoped). Same reactive flow as MVVM, just fewer layers. Edge-case states (loading/empty/error) drive the UI.

Flutter Engine / Dart VM Behavior

Standard (scoped rebuilds; VM logic plain Dart). Not internals-relevant to the round beyond writing efficient rebuilds.

Examples

```
// RIGHT-SIZED (MVVM-lite): thin UI + UI-free view model + simple repo — enough separation, finishable


// data: simple repository (mock or API) — no DTOs/interfaces unless needed
class TodoRepository {
  final _todos = <Todo>[];
```

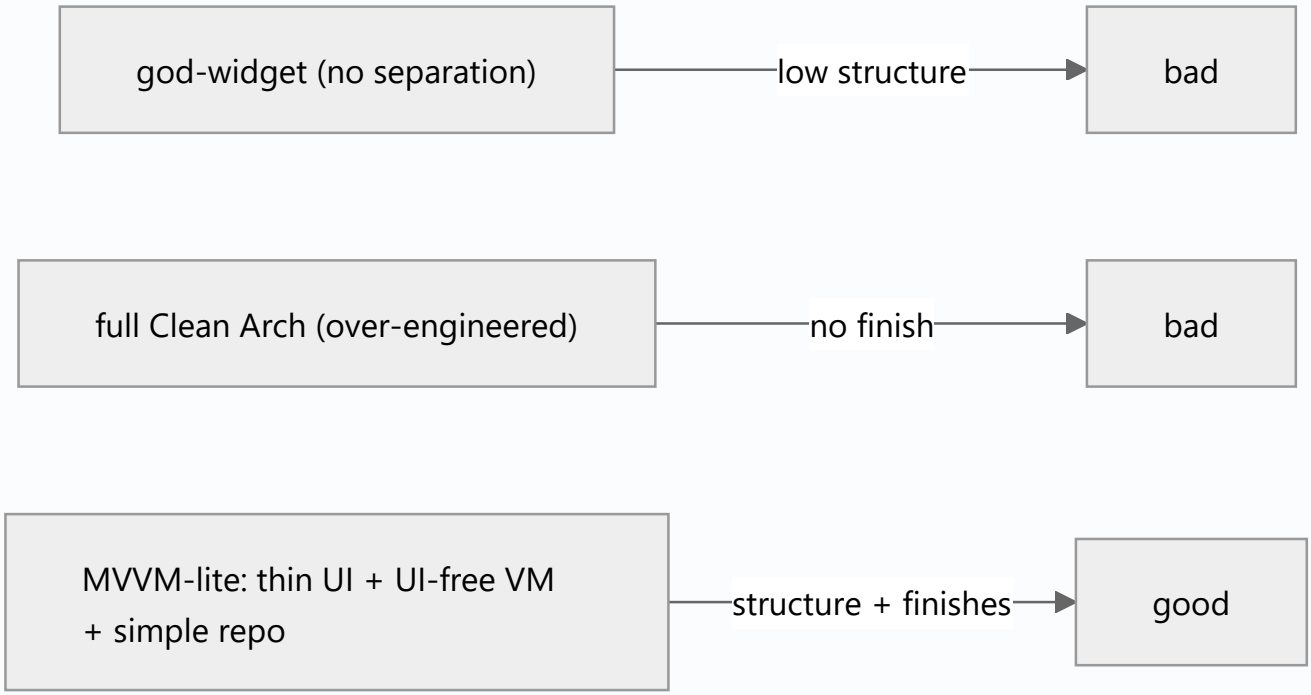
```
  Future<List<Todo>> all() async => List.unmodifiable(_todos);
  Future<void> add(Todo t) async => _todos.add(t);
}

// logic/state: UI-free view model (ChangeNotifier) — state + operations + edge states
class TodoViewModel extends ChangeNotifier {
  final TodoRepository repo;
  TodoViewModel(this.repo);
  TodoState state = const TodoLoading();
  Future<void> load() async {
    state = const TodoLoading(); notifyListeners();
    state = TodoData(await repo.all());              // (add try/catch -> TodoError for edge case)
    notifyListeners();
  }
  Future<void> add(String text) async {
    if (text.trim().isEmpty) return;                 // validation (edge case)
    await repo.add(Todo(text.trim())); await load();
  }
}

// UI: thin widget binds + forwards intents
class TodoScreen extends StatelessWidget {
  const TodoScreen({super.key});
  @override
  Widget build(BuildContext c) {
    final vm = c.watch<TodoViewModel>();
    return switch (vm.state) {
      TodoLoading() => const Center(child: CircularProgressIndicator()),
      TodoData(:final todos) when todos.isEmpty => const Center(child: Text('No todos')),
      TodoData(:final todos) => ListView(children: [for (final t in todos) TodoTile(t)]),
      TodoError() => const Center(child: Text('Something went wrong')),
    };
  }
}
```

```
Right-size by prompt/level:
  trivial (counter/toggle)     -> setState in a well-organized widget
  typical (todo/list+search/form) -> MVVM-lite + ChangeNotifier/Cubit + simple repo   [DEFAULT]
  large/senior take-home       -> + use-case/interface layer + tests (if time)
STOP: no use cases/DTOs/interfaces/DI for a 90-min app (mention "I'd add them if this grew")
```

Diagrams



Common Mistakes

Mistake	Why it scores low	Fix
God-widget (logic/state in `build`)	No separation, unreadable	MVVM-lite: thin UI + UI-free view model
Full Clean Arch (use cases/DTOs/interfaces/DI)	Over-engineered, no time to finish	Right-size: UI/VM/repo only
Picking an unfamiliar state tool under time	Slow, error-prone	Use what you're fluent in (ChangeNotifier/Cubit)
`setState` for shared/async state	Tangled, hard to manage	ChangeNotifier/Provider or Cubit
Deep folder matrix for a tiny app	Navigation overhead	Flat, feature-oriented folders
Business logic in the UI	Untestable/tangled	Logic in the view model
DTOs/mappers for a mock/simple API	Wasted time	Skip unless the API demands it
Edge-case states as ad hoc booleans	Inconsistent UI	Explicit loading/data/empty/error state

Best Practices

- Target **MVVM-lite**: thin **UI** (render + intents) ↔ **UI-free view model** (state + operations + edge-case states) ↔ **simple repository/service** — clear separation with **minimal ceremony**.
- Pick the **simplest state approach that fits** and that you're **fluent in**: `setState` (trivial local), **ChangeNotifier+Provider / Cubit** (shared/async — default); don't adopt an unfamiliar tool under time.
- Keep a **flat, feature-oriented folder structure**; **stop layering at UI/VM/repo** (no use cases/DTOs/interfaces/DI for a ~90-min app) — **mention** you'd add them "if it grew."
- Model **edge cases as explicit states** (loading/empty/error) in the VM; keep the **VM UI-free** (bonus: a quick state-transition **test** if time) — but **finish the feature first**; **right-size to prompt + level**.

Performance

Not runtime perf — the right-sizing *is* the win: enough structure to score, little enough to finish. MVVM-lite keeps the VM testable + UI thin (scoped rebuilds) without the boilerplate that eats your clock. Over/under-structuring both cost you points; the middle finishes clean.

Advantages / Disadvantages

- **+** Scores structure (clear separation, sensible state) *and* finishes (minimal ceremony); testable VM; readable/extensible; fast to build if practiced.
- **−** Requires judgment (where to stop layering), fluency in a state tool, discipline to resist both god-widgets and full Clean Arch under stress.

Interview Questions

1. 🟢 **How much architecture do you use in a machine-coding round?** — Right-sized MVVM-lite: separate thin UI, a UI-free view model (state + ops), and a simple repository — enough to score structure without over-engineering that prevents finishing.
2. 🟢 **Which state management do you pick, and why?** — The simplest that fits and you're fluent in: `setState` for trivial local; ChangeNotifier+Provider or Cubit for shared/async (default) — interviewers care about clean state handling, not a specific library.
3. 🟡 **Where do you stop layering, and why?** — At UI/VM/repo — skip use cases/DTOs/interfaces/DI for a ~90-min app; mention you'd add them if it grew (judgment), but building them wastes finish-time.
4. 🟡 **What are the two opposite failure modes?** — God-widget (no separation → low structure) and full Clean Architecture (over-engineered → no finish); right-sizing avoids both.

5. 🟡 **How do you handle edge cases architecturally?** — As explicit states (loading/empty/error + validation) in the view model, rendered by the UI — scoring both structure and functionality.
6. 🔴 **Why not adopt Bloc/Riverpod if you don't use it daily?** — Under time pressure, an unfamiliar tool slows you and risks errors; use the one you can move fastest in cleanly.
7. 🔴 **How does structure scale with the prompt/level?** — Trivial → `setState` in an organized widget; typical → MVVM-lite; large/senior → add a use-case/interface layer + tests if time.

Senior Engineer Tips

- Default to MVVM-lite (thin UI + UI-free ChangeNotifier/Cubit + simple repo) for almost every prompt; it's the sweet spot that scores structure and lets you finish, and it's fast once practiced.
- Say out loud where you'd add layers "if this grew" while deliberately not building them now; that demonstrates architectural judgment without spending the time that full Clean Architecture would cost.
- Model loading/empty/error as explicit states in the view model and keep it UI-free; you get structure + functionality points cheaply, plus the option of a quick unit test if time remains.

Architect Perspective

Right-sizing under pressure is the handbook's proportionality lesson at its sharpest: MVVM-lite (UI/VM/repo) gives clear, testable separation with minimal ceremony, scoring the structure axis while leaving time to finish — avoiding both the god-widget and full-Clean-Architecture extremes. Choosing a familiar state tool and stopping layering at the right depth (while signaling you *could* go further) is exactly the judgment machine coding rewards, mirroring real-world "scale architecture to the problem" (Module 43, Module 40, Module 11, 02_approach_and_time_management.md).

Summary

- Use MVVM-lite: thin UI ↔ UI-free view model (state + ops + edge states) ↔ simple repository — clear separation, minimal ceremony.
- Pick the simplest state tool you're fluent in (`setState` trivial; ChangeNotifier/Cubit default); flat feature folders; stop layering at UI/VM/repo (no use cases/DTOs/interfaces/DI) — mention you'd add them if it grew.
- Model edge cases as explicit states; keep the VM testable; right-size to prompt/level; finish first, test if time.

Revision Notes

- Target: MVVM-lite — UI (thin: render + intents) ↔ view model (UI-free: state + operations + loading/empty/error states) ↔ simple repository/service (skip DTOs/interfaces/mappers unless API demands). Clear separation, minimal ceremony.
- State: `setState` (trivial local) | ChangeNotifier+Provider / Cubit (shared/async — DEFAULT) | Bloc/Riverpod only if fluent + complex. Use the fastest-clean tool you know; scope rebuilds; VM UI-free/testable.
- Folders: flat, feature-oriented. STOP layering at UI/VM/repo for ~90min app (no use cases/DTO/interfaces/DI — mention "if it grew"). Right-size by prompt/level (trivial→setState; typical→MVVM-lite; large/senior→+use-case/interface+tests). Edge cases as explicit states; quick VM test if time (finish first). Avoid god-widget + over-engineering (opposite failures).

Practice Questions

1. What does MVVM-lite look like, and why is it the sweet spot?
2. Which state approach do you pick for trivial vs shared/async state?
3. Where do you stop layering in a 90-min app, and how do you signal judgment?

Coding Questions

1. Refactor a god-widget prompt into thin UI + UI-free view model + simple repo.
2. Choose + justify a state approach for a given prompt (and reject over-engineering).
3. Model loading/empty/error as explicit states in the view model.

Mini Project

Right-sized structure (prep): For a typical prompt (todo or list+search), implement MVVM-lite: thin UI widgets, a UI-free `ChangeNotifier`/Cubit view model (state + operations + explicit loading/empty/error states + validation), and a simple repository (mock/API) — in a flat feature folder, stopping layering at UI/VM/repo (note where you'd add layers "if it grew"), and add one quick VM unit test if time. Acceptance: clear UI/VM/repo separation (not a god-widget, not full Clean Arch); fluent state tool chosen + justified; edge cases as explicit states; VM UI-free/testable; flat folders; right-sized to prompt/level with a "would add layers if it grew" note.

Common Problems & Patterns

> Most machine-coding prompts are variations on a **small set of recurring problems** — **todo/CRUD list**, **searchable/filterable list** (+ **debounce**), **paginated/infinite-scroll list** (API-backed, loading/error), **form with validation**, **timer/stopwatch**, **counter/cart**, **tabs/master-detail** — each with a **known pattern** you can pre-practice: the data model, the state shape (loading/data/empty/error), the interaction logic (debounce, pagination trigger, validation), and the widget breakdown. Recognizing the prompt → applying the practiced pattern is what lets you **start fast and finish clean** instead of designing from scratch under the clock.

Introduction

This file catalogs the common machine-coding problems + their reusable patterns (state shape, interaction logic, pitfalls), so you can pattern-match a prompt and execute quickly. It applies the right-sized MVVM-lite structure (03_clean_architecture_under_pressure.md) to concrete features.

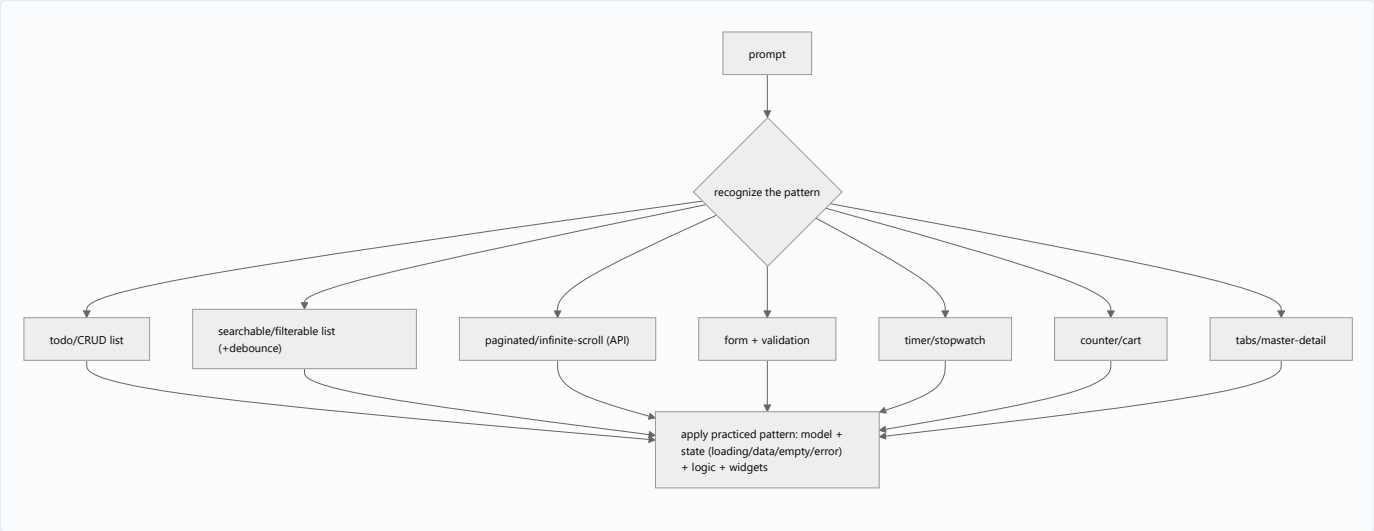
Why this concept exists

Prompts feel varied but reduce to a handful of shapes; **pre-practicing the patterns** means you're not inventing debounce/pagination/validation logic under pressure — you recognize the prompt, reach for the known structure, and spend your time finishing + polishing. It's the machine-coding analog of DSA pattern recognition (Module 55).

Real-world analogy

It's a chef with a repertoire of **base recipes** (a risotto, a reduction, a vinaigrette): a new dish request maps to a known technique they execute smoothly, rather than improvising from nothing. Machine-coding patterns are your **base recipes** — recognize the prompt, apply the practiced method, finish on time.

Internal Working



- **Todo / CRUD list** (the classic):
  - *Model*: `Todo(id, text, done)`. *State*: a list (+ maybe filter: all/active/done). *Logic*: add/toggle/edit/delete (validation: no empty). *Widgets*: input + `ListView.builder` of tiles + filter.
  - *Pitfalls*: no empty-state, no validation, mutating state without notify, no keys on reorderable items.
- **Searchable / filterable list** (+ **debounce**):
  - *State*: full list + query → filtered list (loading if async). *Logic*: filter on query; **debounce** input (~300ms) so you don't filter/fetch per keystroke (Module 43). *Widgets*: search field + results + empty ("no results").
  - *Pitfalls*: no debounce (hammering), no empty-results state, case-sensitivity, not cancelling stale searches.
- **Paginated / infinite-scroll list** (API-backed):
  - *State*: `items + page/cursor + isLoadingMore + hasMore + error`. *Logic*: fetch page 1 (loading), append on scroll-near-bottom (`ScrollController` / `NotificationListener`), stop at `hasMore == false`; **pull-to-refresh**. *Widgets*: `ListView.builder` + a bottom loader/error-retry row.
  - *Pitfalls*: refetching the same page, no loading/error states, no `hasMore` guard, rebuilding the whole list, blocking scroll on error.
- **Form + validation**:
  - *State*: field values + validation errors + submitting. *Logic*: validate (per-field + on submit), disable submit while invalid/submitting, show errors. *Widgets*: `Form` / `TextFormField` (or a VM-driven form) + submit button + error text.
  - *Pitfalls*: no validation, validating only on submit (or never), not disabling submit, losing focus/state, no loading on submit.
- **Timer / stopwatch**:

    - *State*: elapsed + running. *Logic*: `Timer.periodic` (start/pause/reset), format mm:ss, **dispose the timer** on stop/dispose. *Widgets*: display + controls.
    - *Pitfalls*: not disposing the `Timer` (leak/keeps running), rebuilding whole screen each tick, drift (use elapsed from a start time for accuracy).
- **Counter / cart**:
    - *State*: quantities/items + derived total. *Logic*: increment/decrement (guard ≥0), add/remove, compute total. *Widgets*: item rows + total.
    - *Pitfalls*: recomputing incorrectly, negative quantities, not scoping rebuilds (whole list rebuilds on one change).
- **Tabs / master-detail / navigation**:
    - *State*: selected tab/item. *Logic*: switch content; on desktop/web show master-detail side-by-side (responsive — Module 24). *Widgets*: `TabBar` / `NavigationRail` + content.
    - *Pitfalls*: losing tab state, not preserving scroll (use `PageStorageKey` / `AutomaticKeepAlive`), no responsive adaptation if asked.
- **Cross-cutting patterns (apply to all)**:
    - **State shape = loading/data/empty/error** (explicit, in the VM — Module 38) — the single most-forgotten scoring win.
    - **MVVM-lite** structure (thin UI + UI-free VM + simple repo — 03_clean_architecture_under_pressure.md).
    - `ListView.builder` (virtualized) for any list; `const` + scoped rebuilds; **dispose** controllers/timers/subscriptions.
    - **Debounce** for search/typing; `ScrollController` for pagination; **validation** for forms.
    - **Mock the data** if no API is given (an in-memory repo with a fake delay) so you can show loading/error states.
- **Recognize → apply**: on hearing the prompt, **name the pattern(s)** (aloud), reach for the practiced model/state/logic/widgets, and adapt — then spend saved time finishing + polishing + edge cases.

Memory Representation

Not code — a **pattern library**: for each common problem, the (model, state shape, interaction logic, widget breakdown, pitfalls). You pre-build these mentally/by practice so you instantiate them fast under time.

Compiler / Runtime / Engine / VM Behavior

Not applicable beyond running Flutter. Relevant runtime habits: `ListView.builder` virtualization, `Timer` disposal, debounce timers, `ScrollController` for pagination.

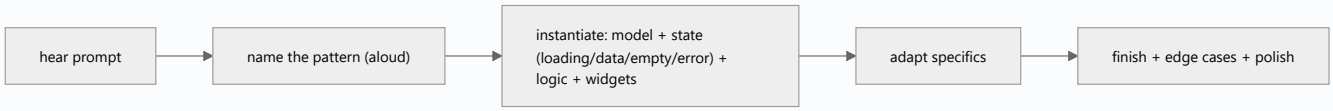
Examples

```
// Searchable list with DEBOUNCE (don't filter/fetch per keystroke) — a recurring pattern
class SearchViewModel extends ChangeNotifier {
  final ProductRepo repo; Timer? _debounce; List<Product> results = [];
  SearchViewModel(this.repo);
  void search(String q) {
    _debounce?.cancel();
    _debounce = Timer(const Duration(milliseconds: 300), () async {   // debounce
      results = await repo.search(q); notifyListeners();
    });
  }
  @override void dispose() { _debounce?.cancel(); super.dispose(); }   // dispose!
}
```

```
// Infinite scroll / pagination pattern — state + trigger + guards
class FeedViewModel extends ChangeNotifier {
  final FeedRepo repo; final items = <Post>[]; int _page = 1;
  bool loadingMore = false, hasMore = true; String? error;
  FeedViewModel(this.repo);
  Future<void> loadMore() async {
    if (loadingMore || !hasMore) return;                 // guard
    loadingMore = true; notifyListeners();
    try {
      final page = await repo.page(_page);
      items.addAll(page.items); hasMore = page.hasMore; _page++;
    } catch (e) { error = 'Failed to load'; }
    loadingMore = false; notifyListeners();
  }
}
// UI: ListView.builder + ScrollController -> loadMore() near bottom; bottom loader/error-retry row.
```

```
// Timer/stopwatch — periodic + dispose + accurate elapsed
class TimerVm extends ChangeNotifier {
  Timer? _t; Duration elapsed = Duration.zero; DateTime? _start;
  void start() { _start = DateTime.now().subtract(elapsed);
    _t = Timer.periodic(const Duration(milliseconds: 100), (_) {
      elapsed = DateTime.now().difference(_start!); notifyListeners(); }); }
  void pause() { _t?.cancel(); }
  void reset() { _t?.cancel(); elapsed = Duration.zero; notifyListeners(); }
  @override void dispose() { _t?.cancel(); super.dispose(); }   // dispose the timer!
}
```

Diagrams



Common Mistakes

Mistake	Where it bites	Fix
No debounce on search	Hammers filter/API	Debounce input (~300ms) + cancel stale
No pagination guards	Refetch/overload	`loadingMore` / `hasMore` guards + trigger near bottom
Missing loading/empty/error states	Every list/async prompt	Explicit state shape in the VM
Not disposing timers/controllers/subs	Timer/search/scroll	`dispose()` everything
Building all list items	Any list	`ListView.builder` (virtualized)
No form validation / no submit-disable	Form prompt	Validate + disable submit while invalid/submitting
Losing tab/scroll state	Tabs/master-detail	`PageStorageKey`/keep-alive
Designing from scratch under time	All	Recognize + apply a practiced pattern

Best Practices

- **Recognize the prompt as a known pattern** (todo/search/paginated/form/timer/cart/tabs) and **instantiate the practiced structure** (model + loading/data/empty/error state + interaction logic + widget breakdown) rather than designing from scratch.
- Apply the **recurring details**: **debounce** search, **pagination guards** (`loadingMore` / `hasMore`) + scroll trigger, **form validation** + submit-disable, `Timer` **+ dispose**, `ListView.builder` + scoped rebuilds, **mock data** with fake delay to show states.

- Always include the **loading/empty/error state shape** (the most-forgotten scoring win) and **dispose** timers/controllers/subscriptions.
- Practice each pattern beforehand so you **start fast + finish clean**; name the pattern aloud (communication) and adapt specifics.

Performance

`ListView.builder` (virtualized), scoped rebuilds, debounce (fewer filter/API calls), and timer disposal keep the built feature efficient — and are also scoring signals. The bigger win is **speed of execution**: practiced patterns let you finish + polish instead of inventing logic under the clock.

Advantages / Disadvantages

- **+** Fast start (pattern recognition), reliable finish, built-in edge cases + best practices, covers most prompts, communicable ("this is the pagination pattern").
- **−** Requires pre-practice of each pattern; must still adapt to the prompt's specifics; over-reliance without understanding fails on twists.

Interview Questions

1. 🟢 **What are the common machine-coding problem types?** — Todo/CRUD list, searchable/filterable list (+debounce), paginated/infinite-scroll (API), form + validation, timer/stopwatch, counter/cart, tabs/master-detail.
2. 🟢 **What state shape should most list/async prompts have?** — Explicit loading/data/empty/error in the view model — the most-forgotten scoring win.
3. 🟡 **How do you implement search correctly?** — Debounce input (~300ms), cancel stale searches, filter/fetch on the debounced query, and show a "no results" empty state.
4. 🟡 **What does the pagination pattern require?** — `items + page/cursor + loadingMore + hasMore + error`, a scroll-near-bottom trigger, guards (don't refetch / stop at `hasMore == false`), pull-to-refresh, and a bottom loader/error-retry row.
5. 🟡 **What must a timer/stopwatch handle?** — `Timer.periodic` for ticks, accurate elapsed from a start time (avoid drift), start/pause/reset, and **disposing the timer** (leak otherwise).
6. 🔴 **Why practice patterns rather than design from scratch?** — Recognition lets you instantiate a known model/state/logic/widget structure fast and spend saved time finishing + polishing + edge cases.
7. 🔴 **What cross-cutting best practices apply to all patterns?** — Loading/empty/error state, MVVM-lite structure, `ListView.builder`, scoped rebuilds, dispose timers/controllers/subs, debounce/validation/mock-data as relevant.

Senior Engineer Tips

- Pre-build the top patterns (todo, search+debounce, pagination, form, timer) until you can produce each in minutes; recognizing the prompt and reaching for the practiced structure is what turns a 90-minute scramble into a finished, polished feature.
- Always add the loading/empty/error state shape and dispose your timers/controllers/subscriptions — those two are the most common cheap points left on the table (and the most common real bugs).
- Mock the data with a fake delay when no API is given, so you can actually demonstrate loading/error states + pagination — a static list hides exactly the behaviors being scored.

Architect Perspective

The pattern library is machine-coding's execution accelerator: recurring problems (list/search/pagination/form/timer) with known model+state+logic+widget structures, all built on right-sized MVVM-lite with explicit edge-case states. Recognizing the prompt and instantiating the practiced pattern is the analog of DSA pattern recognition — it frees your limited time to finish, polish, and communicate, converting knowledge into a consistent, strong submission (03_clean_architecture_under_pressure.md, 02_approach_and_time_management.md, Module 43).

Summary

- Most prompts are variations of: todo/CRUD, searchable list (+debounce), paginated/infinite-scroll, form+validation, timer/stopwatch, counter/cart, tabs/master-detail.
- Each has a practiced pattern (model + loading/data/empty/error state + interaction logic + widget breakdown); recognize → instantiate → adapt → finish.
- Apply recurring details (debounce, pagination guards, validation, `Timer`+dispose, `ListView.builder`, mock data) + always the edge-case state shape.

Revision Notes

- Problems + patterns: todo/CRUD (add/toggle/delete + filter + empty/validation); search (debounce ~300ms + cancel stale + no-results); pagination/infinite-scroll (items+page/cursor+loadingMore+hasMore+error, scroll trigger, guards, pull-to-refresh, bottom loader/error); form (per-field + submit validation, disable submit, loading); timer/stopwatch (`Timer.periodic`, accurate elapsed, start/pause/reset, DISPOSE); counter/cart (guard ≥0 + derived total + scoped rebuild); tabs/master-detail (selected state, keep-alive/PageStorageKey, responsive).
- Cross-cutting: loading/data/empty/error state shape (most-forgotten), MVVM-lite, `ListView.builder` + `const` + scoped rebuilds, dispose timers/controllers/subs, mock data + fake

delay to show states. Recognize→apply (name aloud)→adapt→finish.

Practice Questions

1. What's the state + logic for the pagination pattern?
2. How do you implement search correctly (debounce/edge cases)?
3. What must you always dispose, and what state shape must lists have?

Coding Questions

1. Build a searchable list with debounce + empty-results state.
2. Build an infinite-scroll list with loading/error + `hasMore` guards + pull-to-refresh.
3. Build a stopwatch with start/pause/reset that disposes its timer + avoids drift.

Mini Project

Pattern drill (prep): Pre-build three common patterns (e.g., searchable list with debounce, paginated/infinite-scroll list with loading/error + guards, and a form with validation) using MVVM-lite (UI-free VM + simple mock repo with fake delay), each with the loading/data/empty/error state shape, proper disposal, `ListView.builder`, and the recurring logic (debounce/pagination-guards/validation). Time each build. Acceptance: each pattern implemented with correct state shape + interaction logic + edge cases + disposal; virtualized lists; mock data shows loading/error; built quickly (practiced); pattern named per prompt.

Machine Coding Integration (Capstone: A Worked Round)

> Put it together as a **worked round** for a representative prompt (e.g., "a searchable, paginated list of items with detail"): **clarify + scope** (must-haves vs nice-to-haves + assumptions), **plan** (data model, MVVM-lite structure, state approach, increments), **build incrementally** (working slice first → layer features, keep it running), **handle edge cases** (loading/empty/error/debounce/pagination guards), **reserve time to polish + finish** (+ a quick VM test if possible), all while **communicating** and **watching the clock**. The result: a **finished, clean, right-sized, communicated** feature that scores across all five axes — the repeatable execution that wins machine-coding rounds.

Introduction

This module capstone composes the fundamentals (evaluation axes), the approach (scope→plan→build→edge→polish), right-sized architecture (MVVM-lite), and the pattern library into one end-to-end worked round. It's the "how it all comes together under the clock" deliverable.

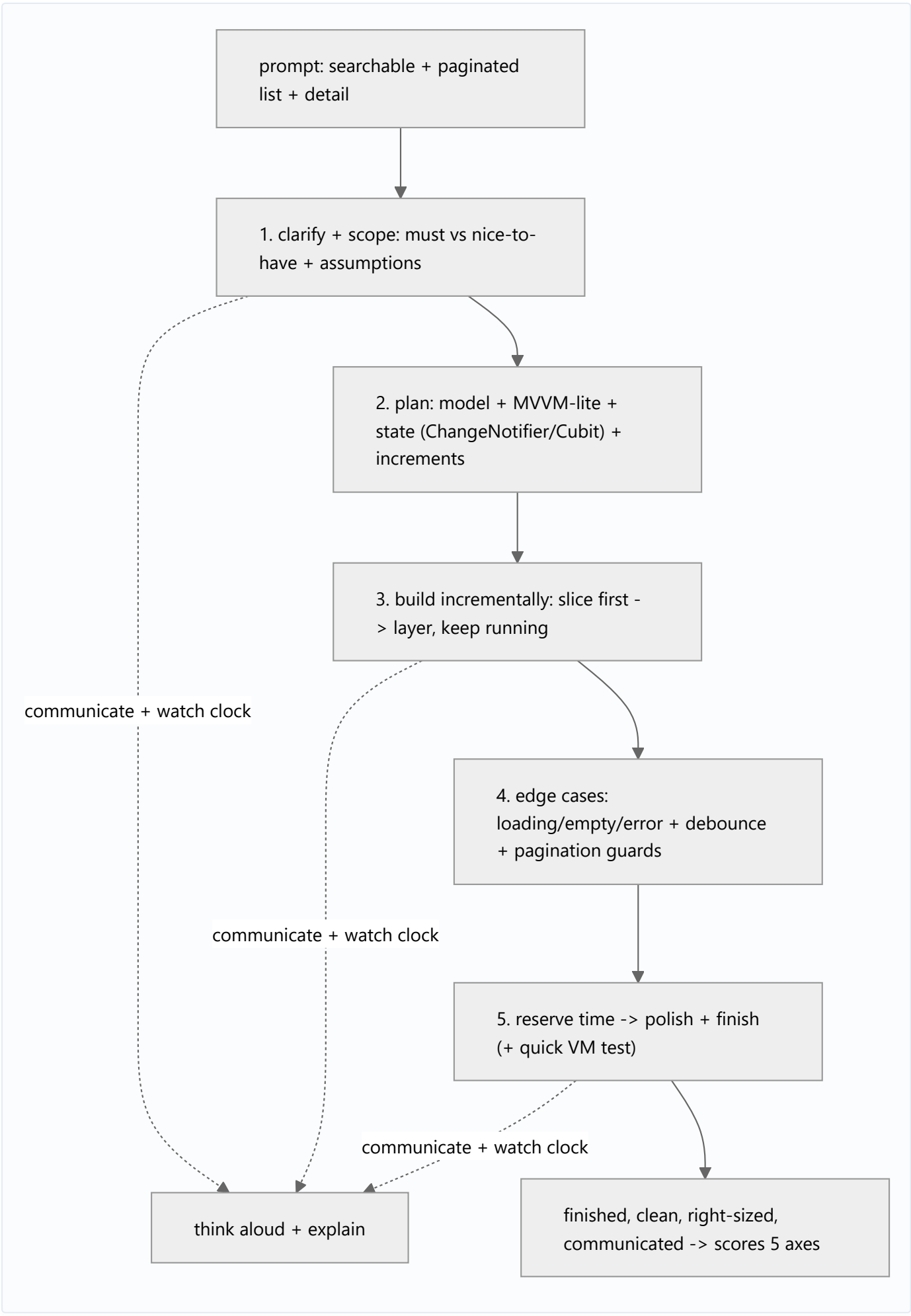
Why this concept exists

The pieces only pay off when **executed together in a single timed run**: pattern recognition + right-sized structure + disciplined time-boxing + communication produce a finished, clean submission. This capstone models that full execution so you can rehearse the whole loop, not just parts.

Real-world analogy

It's a **full timed service run** in the test kitchen: you take the order (clarify), plan the courses (plan), get a plate out fast and keep the line moving (incremental build), taste for problems (edge cases), plate + garnish before time (polish), and narrate to the judges throughout — producing a **complete, well-executed meal on time**, which is exactly what earns the score.

Internal Working


prompt: searchable + paginated list + detail
1. clarify + scope: must vs nice-to-have + assumptions
2. plan: model + MVVM-lite + state (ChangeNotifier/Cubit) + increments
3. build incrementally: slice first -> layer, keep running
4. edge cases: loading/empty/error + debounce + pagination guards
5. reserve time -> polish + finish (+ quick VM test)
finished, clean, right-sized, communicated -> scores 5 axes
think aloud + explain
communicate + watch clock
communicate + watch clock
communicate + watch clock

- **Prompt (representative)**: "Build a screen listing items from an API with **search** and **infinite-scroll pagination**; tapping an item opens a **detail** screen." Combines the search + pagination + navigation patterns (04_common_problems_and_patterns.md).
- **1. Clarify + scope** (~5–8 min) (02_approach_and_time_management.md): confirm data source (assume a mock/provided API), page size, search behavior; **must-haves** = list + search + pagination + loading/error + detail; **nice-to-haves** = pull-to-refresh, caching, animations. State assumptions aloud.
- **2. Plan** (03_clean_architecture_under_pressure.md): **model** `Item`; **MVVM-lite** — `ItemRepository` (mock/API, `search` / `page`), `ItemListViewModel` (`ChangeNotifier`: state = items + query + loadingMore + hasMore + error; ops = `search` (debounced), `loadMore`), thin `ItemListScreen` + `ItemTile` + `ItemDetailScreen`; **increments**: static list → data → loading/empty/error → search(debounce) → pagination → detail nav → polish.
- **3. Build incrementally**: get the **list rendering fast** (static → mock data), keep it **running** each step; layer **loading/empty/error**, then **debounced search**, then **infinite-scroll** (`ScrollController` + `hasMore` / `loadingMore` guards), then **detail navigation**. Core first; app demoable throughout.
- **4. Edge cases** (Module 38): loading spinner, empty ("no results"), error + retry, **debounce** (~300ms, cancel stale), **pagination guards** (no refetch, stop at `hasMore == false`), input trimming — the scored functionality details.
- **5. Polish + finish + (test)**: reserve the last ~10–15% — tidy naming/dead code, ensure UI is reasonable, **self-run** all states, and (if time) a **quick VM unit test** (e.g., "search updates results", "loadMore appends + respects hasMore") — the UI-free VM makes this cheap (Module 49). **Nothing broken.**
- **Communicate throughout**: announce the plan + increments, explain the state choice + debounce/pagination decisions, state what you're cutting (nice-to-haves) if behind, and note "I'd add caching/DTOs/tests if this grew" (judgment) — scored heavily.
- **Time-box** (~90 min): ~8 clarify/plan, ~55 build core (list→data→states→search→pagination→detail), ~12 edge cases, ~15 polish/finish(+test); **cut nice-to-haves, not correctness/finishing** if behind; **don't rabbit-hole** (stub + move on).
- **Result = all five axes**: **functionality** (works + edge cases), **code quality** (readable MVVM-lite), **structure** (UI/VM/repo separation + sensible state), **completeness** (finished + polished), **communication** (throughout) — a strong, offer-winning submission (01_machine_coding_fundamentals.md).
- **Right-size to level**: junior → list + basic search + states; mid → + pagination + clean MVVM-lite + edge cases; senior → + detail + a test + trade-off narration (+ mention scaling).

Memory Representation

Not new structures — the **executed plan**: model + MVVM-lite (UI-free VM with the search/pagination state shape + a mock repo) built in running increments, plus a time-box + communication track. The submission is a finished, clean, demoable feature.

Compiler / Runtime Behavior

Kept compiling/running at each increment (hot reload); the VM compiles UI-free (testable); `ListView.builder` + `ScrollController` drive virtualized, paginated rendering; debounce timers cancelled on dispose.

Flutter Engine / Dart VM Behavior

Standard: virtualized list rendering, scoped rebuilds on state changes; VM logic plain Dart (fast optional test). Nothing internals-specific beyond good rebuild habits.

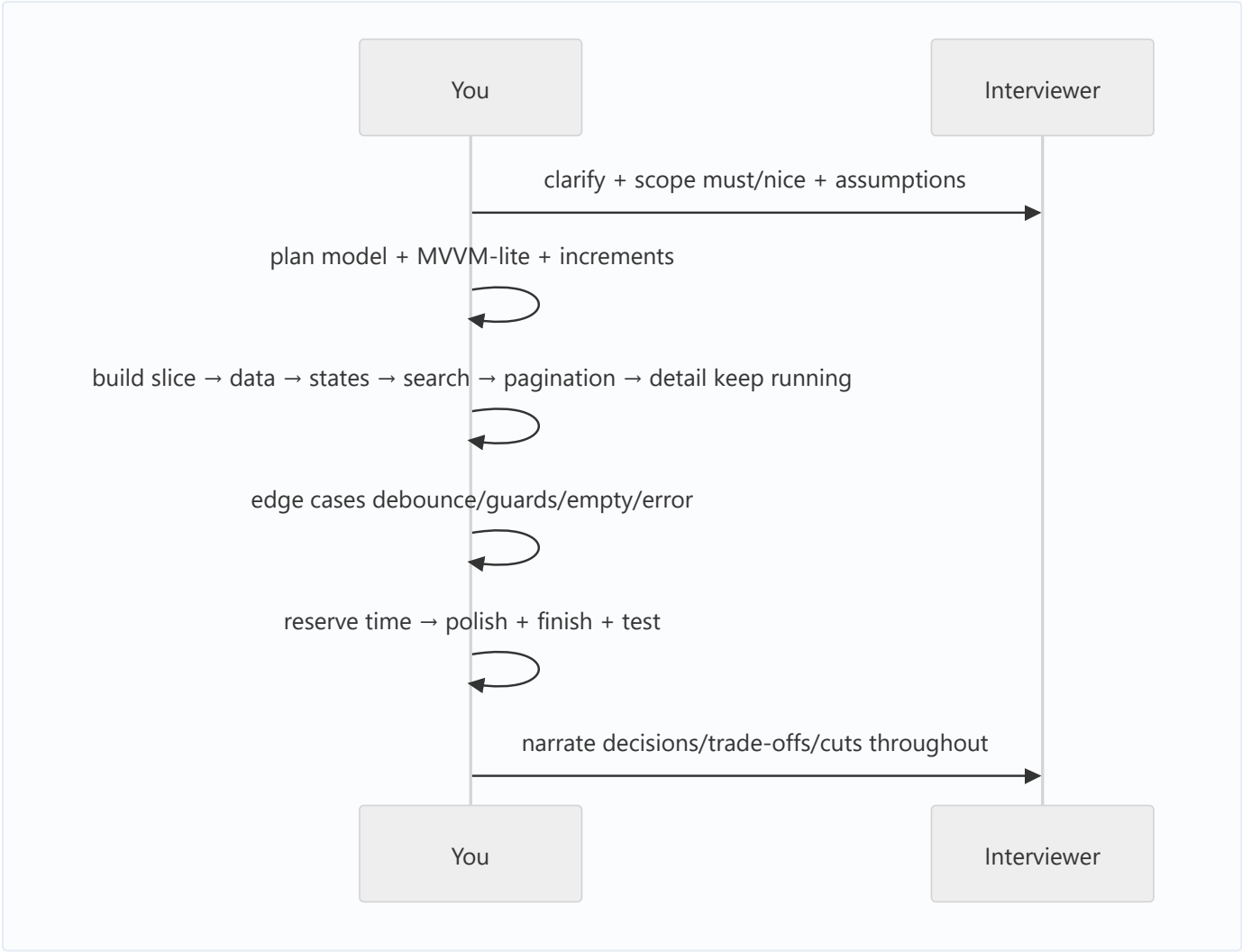
Examples

```
// The composed view model (MVVM-lite): search (debounce) + pagination (guards) + states
class ItemListViewModel extends ChangeNotifier {
  final ItemRepository repo; Timer? _debounce;
  final items = <Item>[]; String query = ''; int _page = 1;
  bool loading = true, loadingMore = false, hasMore = true; String? error;
  ItemListViewModel(this.repo) { _load(reset: true); }

  void search(String q) {                          // debounced search
    query = q; _debounce?.cancel();
    _debounce = Timer(const Duration(milliseconds: 300), () => _load(reset: true));
  }
  Future<void> loadMore() async {                  // pagination (guards)
    if (loadingMore || !hasMore || loading) return;
    loadingMore = true; notifyListeners(); await _fetch(); loadingMore = false; notifyListeners();
  }
  Future<void> _load({bool reset = false}) async {
    if (reset) { _page = 1; hasMore = true; items.clear(); loading = true; error = null; notifyListeners();
}
    await _fetch(); loading = false; notifyListeners();
  }
  Future<void> _fetch() async {
    try { final p = await repo.page(query, _page); items.addAll(p.items); hasMore = p.hasMore; _page++; }
    catch (_) { error = 'Failed to load'; }
  }
  @override void dispose() { _debounce?.cancel(); super.dispose(); }   // dispose!
}
// UI: thin screen binds to state -> loading/empty/error/data; ScrollController -> loadMore near bottom;
// tile onTap -> Navigator.push(ItemDetailScreen). (add try/catch UI states + retry.)
```

```
Time box (~90 min) worked:
  0-8    clarify + scope (must/nice-to-have + assumptions) + plan (model/MVVM-lite/state/increments)
  8-63   build core: static list -> mock data -> loading/empty/error -> debounced search -> pagination ->
detail nav
  63-75  edge cases (debounce cancel, pagination guards, empty/error/retry, input trim)
  75-90  polish + finish (naming/dead code/UI, self-run all states) + quick VM test if time
  throughout: communicate plan/decisions/cuts; cut nice-to-haves (not correctness/finish) if behind
```

Diagrams



Common Mistakes

Mistake	Why it fails	Fix
Skipping clarify/scope	Builds wrong/over-scoped	Clarify + must/nice-to-have + assumptions first
No plan / god-widget	Thrash + low structure	Plan MVVM-lite + increments
Over-engineering (full Clean Arch)	No time to finish	Right-size (UI/VM/repo)
Gold-plating before core	No finish	Core first (list→states→search→pagination)
Missing edge cases (loading/empty/error/debounce/guards)	Fails functionality	Add them explicitly
No reserved polish/finish	Broken at buzzer	Reserve last ~15%
Rabbit-holing pagination detail	Blows budget	Stub/simplify + move on
Silent execution	Misses communication axis	Narrate throughout

Best Practices

- Execute the **full loop**: clarify + scope (must/nice-to-have + assumptions) → plan (model + MVVM-lite + state + increments) → build incrementally (working slice first, keep running, core first) → edge cases (loading/empty/error + debounce + pagination guards) → **reserve time to polish + finish** (+ quick VM test if possible).
- Use **right-sized MVVM-lite** (UI-free VM with the search/pagination state shape + a mock repo) and the **practiced patterns** (debounce, pagination guards, states) — recognize + instantiate, don't design from scratch.
- **Time-box aggressively** (~8/55/12/15), **cut nice-to-haves (not correctness/finishing)** if behind, **don't rabbit-hole**, and **communicate throughout** (plan/decisions/cuts/"would-add-if-grew").
- Aim to score **all five axes** (functionality/quality/structure/completeness/communication); **right-size to level**.

Performance

Not runtime perf — the "performance" is a finished, clean, five-axis-scoring submission within the box. The composed approach + patterns + right-sizing maximize it: fast start (recognized patterns), always-demoable (increments), scored details (edge cases + communication), polished finish (reserved time).

Advantages / Disadvantages

- **+** Repeatable, high-scoring execution (all five axes), always-demoable, finished + clean + communicated, right-sized, level-calibrated.
- **−** Requires practiced patterns + disciplined time-boxing + communication under stress; must resist over-engineering/gold-plating/rabbit-holes.

Interview Questions

1. 🟢 **Walk through a machine-coding round end-to-end.** — Clarify + scope (must/nice-to-have + assumptions) → plan (model + MVVM-lite + state + increments) → build incrementally (slice first, keep running, core first) → edge cases → reserve time to polish + finish (+ test), communicating throughout.
2. 🟢 **What architecture + state do you use for a searchable/paginated list?** — MVVM-lite: thin UI + a UI-free `ChangeNotifier`/Cubit view model (items/query/loadingMore/hasMore/error + debounced search + guarded loadMore) + a simple/mock repository.
3. 🟡 **How do you time-box a ~90-min round?** — ~8 clarify/plan, ~55 build core, ~12 edge cases, ~15 polish/finish(+test); cut nice-to-haves (not correctness/finishing) if behind; don't

rabbit-hole.

4. 🟡 **What edge cases must this prompt handle?** — Loading/empty/error + retry, debounce (cancel stale), pagination guards (no refetch, stop at `hasMore == false`), input trimming.
5. 🟡 **How do you demonstrate judgment on scope/architecture?** — Right-size (MVVM-lite, no full Clean Arch), state what you're cutting, and note "I'd add caching/DTOs/tests if this grew" — building only what fits the box.
6. 🔴 **How do you ensure you finish?** — Working slice early + incremental core-first build (always running) + reserved polish time + cutting nice-to-haves before correctness — never a broken half-feature at the buzzer.
7. 🔴 **How does the submission score across the five axes?** — Functionality (works + edge cases), code quality (readable MVVM-lite), structure (UI/VM/repo + sensible state), completeness (finished/polished), communication (throughout).

Senior Engineer Tips

- Rehearse the whole loop on a couple of representative prompts (searchable+paginated list, todo) until clarify→plan→incremental-build→edge-cases→polish is automatic; the round is won by smooth execution, not novel ideas.
- Keep the app running after every increment and reserve the last ~15% for polish + a quick VM test; a finished, clean, five-axis submission beats an ambitious broken one every time.
- Narrate decisions and cuts, and explicitly say what you'd add "if this grew"; that combination of finishing cleanly + communicating judgment is exactly the senior signal machine coding is looking for.

Architect Perspective

The worked round is the synthesis of machine-coding skill: pattern recognition + right-sized MVVM-lite + disciplined time-boxing + communication, executed to a finished, clean, five-axis submission. It mirrors real engineering under constraints — scope, prioritize, build incrementally, handle edge cases, finish, communicate — which is exactly why it predicts on-the-job ability. Rehearsed until automatic, it converts the handbook's knowledge into a consistent, offer-winning performance (01_machine_coding_fundamentals.md, 02_approach_and_time_management.md, 03_clean_architecture_under_pressure.md, 04_common_problems_and_patterns.md).

Summary

- A worked round: clarify + scope → plan (model + MVVM-lite + state + increments) → build incrementally (slice first, keep running, core first) → edge cases (loading/empty/error + debounce + pagination guards) → reserve time to polish + finish (+ quick test), communicating throughout.

- Use right-sized MVVM-lite + practiced patterns; time-box aggressively; cut nice-to-haves not correctness/finishing; don't rabbit-hole.
- Produce a finished, clean, right-sized, communicated feature scoring all five axes; right-size to level.

Revision Notes

- Worked loop: clarify+scope (must/nice-to-have + assumptions, ~8min) → plan (model + MVVM-lite: repo/VM/UI + state approach + increments) → build incrementally (static list→data→loading/empty/error→debounced search→pagination guards→detail nav, keep running, core first) → edge cases → reserve ~15% polish+finish (+quick VM test) → communicate throughout.
- VM state shape: items/query/loading/loadingMore/hasMore/error; debounced search (cancel/dispose); loadMore guards; UI-free (testable). Time-box ~8/55/12/15; cut nice-to-haves not correctness/finish; no rabbit-holes; right-size to level. Scores all five axes (functionality/quality/structure/completeness/communication).

Practice Questions

1. Walk the full worked round for a searchable/paginated list.
2. What's the VM state shape + logic, and how do you keep it testable?
3. How do you time-box + what do you cut if behind?

Coding Questions

1. Build the searchable/paginated list + detail (MVVM-lite) end-to-end with edge cases.
2. Add a quick VM unit test (search updates results / loadMore respects hasMore).
3. Time yourself and reflect against the five evaluation axes.

Mini Project

Full worked round (capstone — prep): Do a complete timed (~90-min) machine-coding run for a representative prompt (searchable + paginated list + detail): clarify + scope (must/nice-to-have + assumptions), plan (model + MVVM-lite + state + increments), build incrementally (keep running, core first), handle edge cases (loading/empty/error + debounce + pagination guards), reserve time to polish + finish (+ a quick VM test), narrating throughout — then reflect against the five axes. Acceptance: clarified + scoped + planned; incremental build (always running, core first); MVVM-lite (UI-free VM with the search/pagination state shape + mock repo); edge cases handled; finished + polished within the box; quick VM test if time; communicated; self-scored on all five axes; right-sized to target level.