
SWENT - SoftWare EnTerprise

BA5 • 2026-2027

Contents

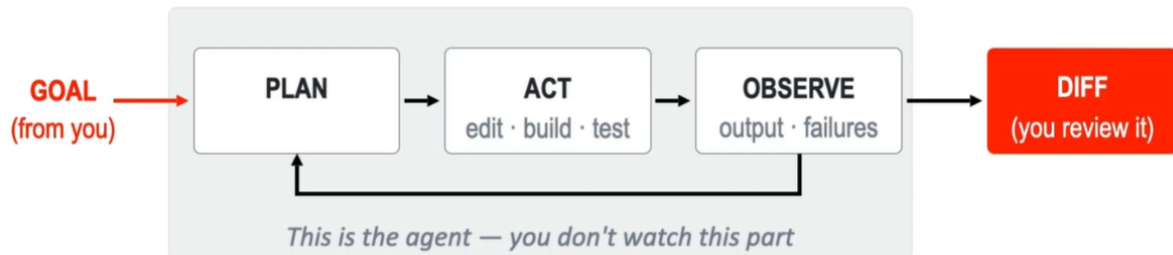
1	Week1 : Part A	4
1.1	Agentic Software Development	4
1.2	Requirements	5
1.3	User Story	6
1.4	Validating & Prioritizing Requirements	7
1.5	Epics	7
2	Week1 : Part B	9
2.1	Mobile Platforms	9
2.2	The Mobile Device	9
2.2.1	The Mobile SoC = A "Committee" of Processors	9
2.2.2	Fragmentation	10
2.3	Mobile OS vs Server OS	10
2.3.1	User Model and Security	11
2.3.2	Power Management	11
2.3.3	Execution Model & App Lifecycle	11
2.3.4	Network & Connectivity	11
2.3.5	UI & Input/Ouput	12
2.3.6	Storage & Filesystem	12
2.3.7	Hardware Environment	12
2.3.8	Software Updates & Maintenance	12
2.3.9	Difference between Android/IOS	13
2.4	App Data (Persistence Levels)	13
2.5	The Split-App Model	14
2.5.1	Benefits of Split-App Model	14
2.5.2	Backend/Network Services	14
2.5.3	Push Notifications	15
2.6	MVVM	15
2.6.1	Benefits of MVVM	16
2.7	Components of A Mobile App	17
2.7.1	Four Android App Components	17
2.7.2	Activities	17
2.7.3	Inter-App Communication via Intents	17
2.8	The Lifecycle of an Android Activity	18

3	Week2 : Part A	21
3.1	Testing: Different Levels	21
3.1.1	Writing Good Unit Tests	21
3.2	The Cost of Bugs	21
3.3	How Well Can We Test?	22
3.4	Coverage Metrics	22
3.5	Technical Debt	23
3.6	Code Smells	23
3.7	What Coverage Doesn't Catch	24
3.7.1	The Imperfection of Coverage	24
3.7.2	Smartphone Platform: Heterogeneity Challenges to Testing	24
3.7.3	Testing Challenges: Non-Functional Requirements	24
3.7.4	Careful What You Ask For	24
3.8	Takeaways	25
4	Week2 : Part B	26
4.1	Test Doubles Mocks, Stubs, Fakes	26
4.1.1	Testing a ViewModel That Talks to the Network	26
4.1.2	Breaking the Dependency	26
4.1.3	Test Doubles	27
4.2	Dependency Injection	27
4.3	Regression Testing	28
4.3.1	Building a Regression Suite	29
4.3.2	Regression Test Suite: Summary	29
4.4	UI Testing for Your App	30
4.4.1	Where Does Validation Belong?	30
4.4.2	What to Test with UI Tests	30
4.4.3	UI Testing: A Recap	30
5	Week3 :	31
5.1	Scrum	31
5.2	Continuous Integration / Continuous Delivery & Deployment	31
5.2.1	The CI/CD Pipeline	32
5.2.2	Our Pipeline for the Bootcamp	33

CHAPTER 1

Week1 : Part A

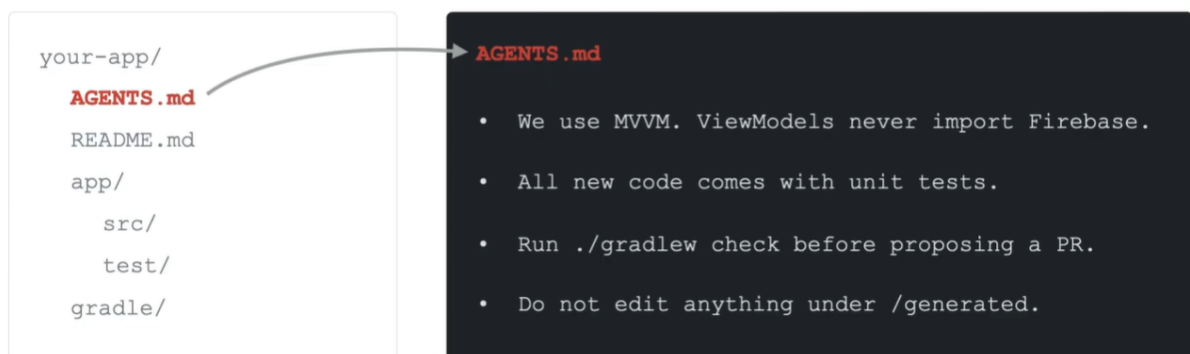
1.1 Agentic Software Development



What the agent does not know

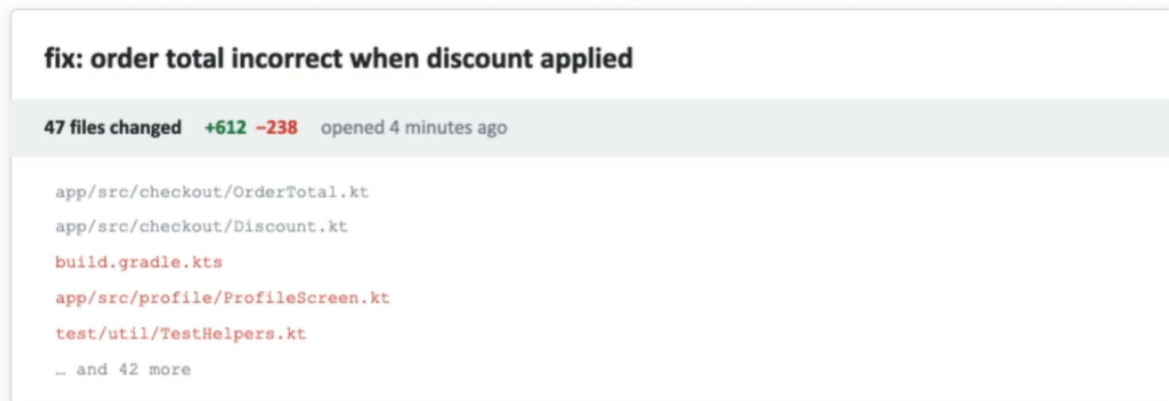
- **Context** — why the codebase is the way it is (the auth layer looks strange for a reason).
- **Decomposition** — turning “build the food ordering feature” into one reviewable change.
- **Acceptance criteria** — what “done” means; if you do not say, it decides optimistically.
- **Permissions** — what it may touch, and what must be approved by you first.

Only you have these — the agentic loop is blind to them.



Write down the rules for the agent (it will help your team agree on them)

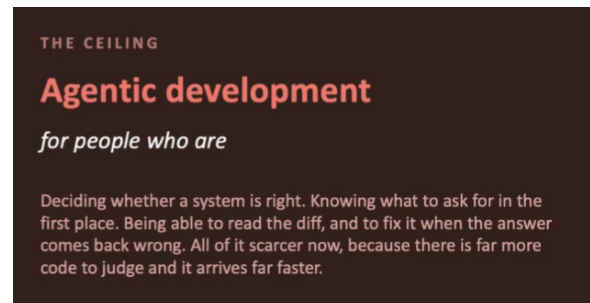
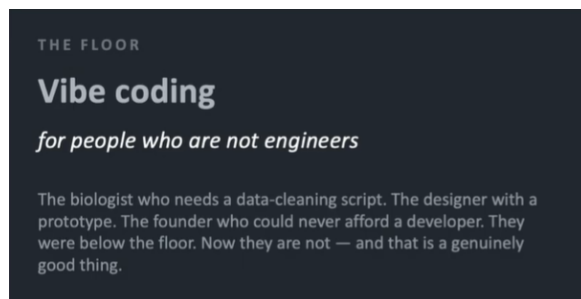
What if the agent opened this PR?



Answer with : *"I can't review this. Please split it: the total calculation in one PR, the config change in another, and drop all unrelated screen edits."*

How agentic can go wrong

- **Confidently wrong at scale**
"refactored 30 files and broke one thing in each"
- **Hallucinated dependencies**
"imported a library that doesn't exist, and added one that shouldn't"
- **Tests that pass and check nothing**
"wrote tests and coverage went up, but correctness went down"
- **Plausible architecture drift**
"every feature is fine in isolation, but nothing fits together anymore"



1.2 Requirements

Listen relentlessly · Listen for the need behind the want

How do we define requirements?

- Most users are not precise and objective.
- Elicit requirements by talking & listening to (potential) users.
- Group users in equivalence classes around personas.
- Develop user stories.

Can I use an LLM instead of real users?

- **Not for the persona, and not for the need**
 - *Five user stories in ten seconds — fluent, plausible, well-structured.*
 - *But nobody said any of it.*
 - *This is the faster-horses failure, with better grammar.*
- **Yes, once a real person has told you something**
 - *Turn a vague sentence into a testable one.*
 - *Propose the edge cases you did not think of.*
 - *Draft acceptance criteria.*

Use an LLM to sharpen a requirement you got from a person — not to invent the person.

1.3 User Story**User story template**

As a	[role],	<i>who is the primary stakeholder?</i>
I want to	[action],	<i>what effect does she want the story to have?</i>
so that	[reason].	<i>what value will she derive from that effect?</i>

Example. As a **SwEnt student**, I want to **learn about software requirements**, so that **I can engineer better software**.

Warning

- If you can only fill the template with a very generic role or reason, the feature is probably not something a real user actually cares about.
- The developer is not an end user — “as a developer” is not a valid role in a user story.
- If the story dictates *how* it should be implemented, it is too specific: a user story states the problem, not the solution.

Good user stories: INVEST

- **I**ndependent — *self-contained; can be built without waiting on other stories*
- **N**egotiable — *captures the problem, not the solution, leaving room for the implementor*
- **V**aluable — *delivers value the end user can see*
- **E**stimable — *small enough that the effort can be estimated*
- **S**mall — *fits inside a single sprint*
- **T**estable — *comes with clear acceptance criteria*

What to do about shared needs?

- **Truly universal desire** — one story with a deliberately generic role.

As a user, I want to reset my password, so that I can regain access to my account.

- **Capture fine distinctions in separate user stories.**

As a student or teacher, I want to access the course materials on Moodle, so that I can use them for studying or teaching, respectively.

- **Persona groups (“meta-persona”)** — one role standing for several personas that share the need.

As a member of the IT department. . .

As a mobile app user. . .

1.4 Validating & Prioritizing Requirements

Validating requirements: are they right?

- **Is your software really answering users’ needs and wants?**
 - *use prototypes to elicit feedback*
 - *validate “in the wild”*
- **The outcome of validation is typically not true/false** — it is a matter of degree, not a binary verdict.

Prioritizing requirements

- **Importance of features varies constantly.**
- **Common prioritization schemes**
 - *P0 / P1 / P2*
 - *MoSCoW: must-have, should-have, could-have, won’t-have*
- **Prioritization is primarily driven by value to the user**
 - *but also weighs technical feasibility, legality, cost, . . .*

1.5 Epics

Epic

An **epic** is a broad set of features that spans **multiple user stories**.

- Captures larger, more complex user needs.
- Enables a comprehensive understanding of a feature set.
 - *once divided into actionable, specific user stories, the big picture can get lost*
- Useful for organizing work.
 - *ensures the development process stays aligned with user needs*
- A **user story** then becomes a specific, actionable item that contributes to the broader goal of the epic.

Example. Epic: *Personalized and Streamlined Dining Experience.*

As a user of PolyFood, I want the app to understand my habits, preferences, and context, so that ordering food becomes faster, smarter, and more enjoyable over time.

User flow vs. epic

- **User flow** — a specific sequence of steps a user takes to accomplish one task in the app, from start to finish.
 - **Login flow**: *launch screen → login screen → error handling for wrong credentials → main dashboard on success*
- **Epic** — a broader, high-level feature or goal that usually spans several user flows or smaller tasks.
 - **Account Management epic**: *login flow + password-reset flow + account-creation flow + edit-profile flow*

Recipe for user-centric software

1. Elicit user requirements.
2. Formulate user stories.
 - *should be **I**ndependent, **N**egotiable, **V**aluable, **E**stimable, **S**mall, and **T**estable*
3. Validate and prioritize them.
 - *all stakeholders must be involved*
4. Organize user stories into epics.
5. Plan the work.

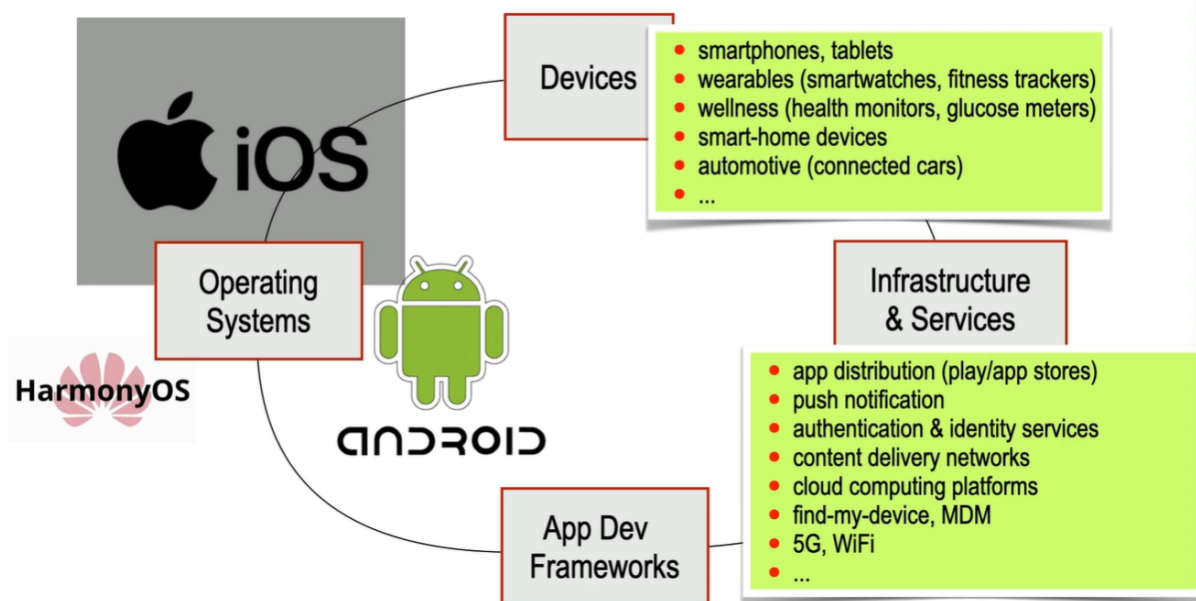
CHAPTER 2

Week1 : Part B

2.1 Mobile Platforms

Mobile platform

The ingredients that come together to enable the **creation**, **distribution**, and **operation** of mobile content and services.

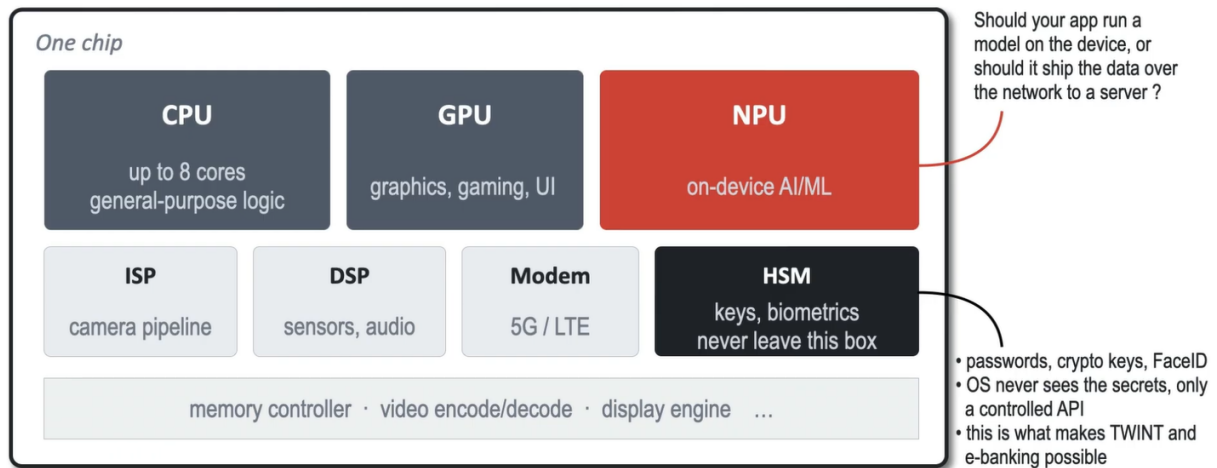


2.2 The Mobile Device

2.2.1 The Mobile SoC = A "Committee" of Processors

Mobile SoC (System-on-Chip)

A single chip integrating many specialized processors, each optimized for a different job. This heterogeneity is what lets a phone deliver serious compute for a few Watts.

**On-device vs. server**

The NPU raises a design question for every app that uses ML: should the model run on the device, or should the data be shipped over the network to a server?

Why the HSM matters

- Stores passwords, crypto keys, FaceID data.
- The OS never sees the secrets, only a controlled API.
- This isolation is what makes TWINT and e-banking possible.

2.2.2 Fragmentation

24,000+ distinct Android device models in active use

- Shape** Foldables have gone mainstream. Your app needs to accommodate screens from 3" to 10", and on foldables the size can change while the app is running.
- Age** There are 7 different Android versions active in the market => the phone in your pocket is not the phone your users own.
- Capability** NPU performance varies significantly by tier => on-device AI runs beautifully on a flagship phone and not at all on a mid-range phone.

Warning

How do you make your app work across the board?

2.3 Mobile OS vs Server OS

2.3.1 User Model and Security

Feature	Mobile OS	Server OS
User model	Single-user	Multi-user with roles/permissions
App/process isolation	App sandboxing	Process-level isolation (UIDs/GIDs)
Permission system	Explicit runtime permissions	Access control via file and system permissions
Admin/root access	Not available to user	Sudo/admin access available
Trust model	Distrusts apps	Distrusts users

2.3.2 Power Management

Feature	Mobile OS	Server OS
Power	Battery-powered, energy-constrained	Always-on power
Background processing	Throttled/deferred	Unrestricted (daemons, services)
Thermal limits	Aggressively managed	Passive or active cooling, less constrained

2.3.3 Execution Model & App Lifecycle

Feature	Mobile OS	Server OS
Process lifecycle	OS can kill/restart apps anytime	Processes expected to run indefinitely
Lifecycle control	App lifecycle events (onPause, etc.)	None; processes control their own lifetime
Scheduling priority	UI threads get CPU priority	Throughput-optimized scheduling

2.3.4 Network & Connectivity

Feature	Mobile OS	Server OS
Network stability	Intermittent or mobile	Stable, high-bandwidth
Network types	Cellular, WiFi, Bluetooth, NFC	Ethernet, bonded interfaces, high-speed
Network policy	Battery-sensitive, may block traffic	No such constraints

2.3.5 UI & Input/Output

Feature	Mobile OS	Server OS
User interaction	Touch, gestures, voice, sensors	None (headless); CLI or remote GUI
UI importance	Must be real-time and smooth	Not applicable
Output interfaces	Small screens, haptics, speakers	Logs, stdout, APIs, remote outputs

2.3.6 Storage & Filesystem

Feature	Mobile OS	Server OS
Storage type	Flash (UFS), constrained size	Large-scale SSD/HDD, RAID
File access	Per-app private storage	Shared/global filesystem
Access control	App permission-based	User/group permissions

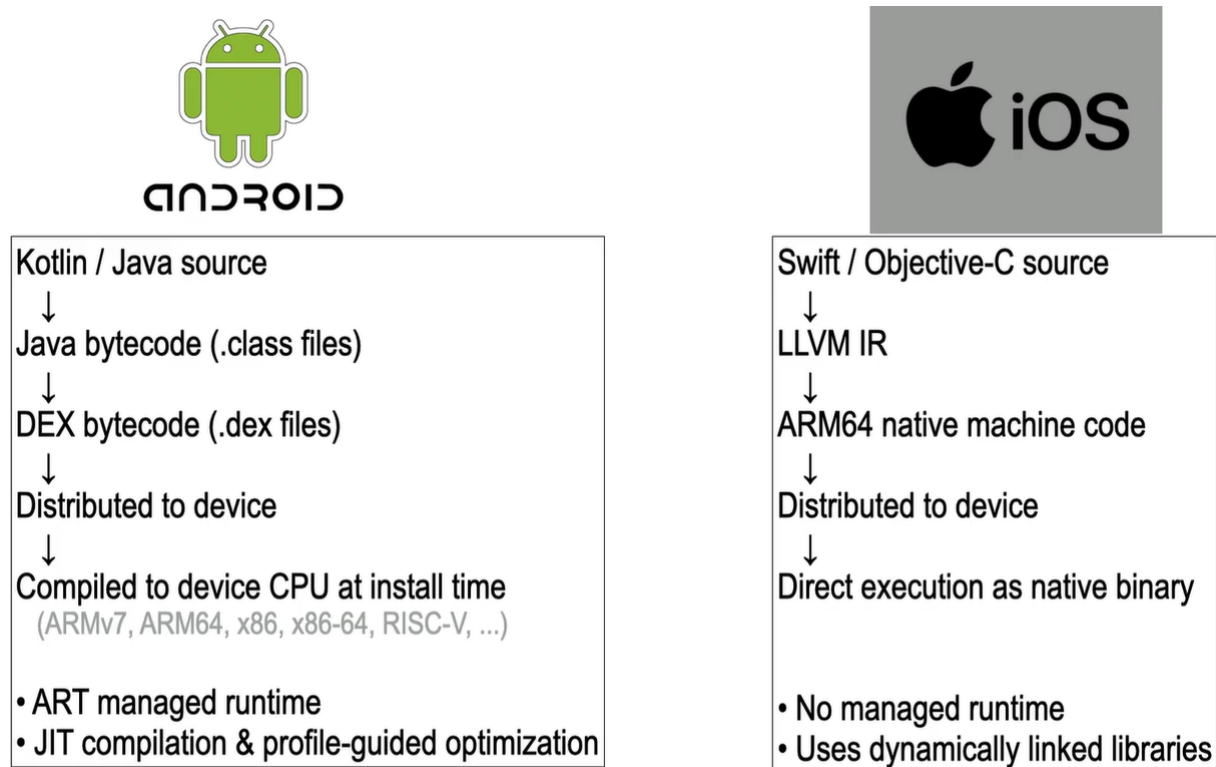
2.3.7 Hardware Environment

Feature	Mobile OS	Server OS
Architecture diversity	Extreme (ARM SoCs, sensors, NPUs)	Standardized (x86_64 or server-grade ARM)
Input/output diversity	Cameras, touchscreens, GPS, accelerometers	Basic I/O, remote-managed
Integration level	Highly integrated (SoC = CPU+GPU+NPU)	Modular hardware, discrete components

2.3.8 Software Updates & Maintenance

Feature	Mobile OS	Server OS
Update mechanism	OTA updates via OEM/carrier	Admin- or automation-driven updates
Control over updates	Limited (esp. on Android)	Full control (scripts, policies)
Update timing	Varies by vendor/device	Regular, scheduled maintenance

2.3.9 Difference between Android/iOS



2.4 App Data (Persistence Levels)

Service	Usage examples	Lifespan	Implementation
SharedPreferences	user prefs (dark mode, toggles), simple game high scores	Until app is uninstalled or data is cleared	Key-value pairs, lightweight
SQLite / Jetpack Room	draft emails, chat history, notes, complex leaderboards	Until app is uninstalled or data is cleared	Structured relational DB
Internal storage (files)	app-specific documents, private media, email attachments	Until app is uninstalled or data is cleared	Private to the app
External storage (files)	shared media (photos, audio, video), downloads	Until user deletes it or external storage is wiped	Accessible to other apps (with permission)
Cache directories	cached images, temporary API responses	May be cleared anytime by OS or user	For temporary but re-creatable data
Content providers	contacts, calendar events, media store entries	Until user deletes (system-level state)	Lets apps read/write shared system data
Cloud / backend services	synced drafts, saved games, messaging data	Persists across devices, as long as cloud account exists	Not local; depends on server availability

Which row survives what?

Say your app remembers the user's favorite restaurants.

- Which row should it use to survive a **reinstall of the app**?
- Which row should it use to survive a reinstall on a **new phone**?

Answer

- **Survives reinstall (same phone):** SharedPreferences, Room and internal storage all die on uninstall. External storage and cloud storage both survive.
- **Survives reinstall on a new phone:** external storage won't work, since it never existed on that phone in the first place. Only **cloud storage** works.

Choose the storage service based on data complexity, visibility, and required lifespan – a real project will typically combine more than one row of this table.

2.5 The Split-App Model

- **Client-side (frontend)**
 - runs on the user's device, provides the UI and UX
- **Server-side (backend)**
 - runs on a server or in the cloud
- **Client interacts with cloud services**
 - typically REST APIs or GraphQL
 - send HTTP request
 - get back JSON or XML

2.5.1 Benefits of Split-App Model

Property

Modularity + contracts between the frontend and backend $\Rightarrow$ highly decoupled $\Rightarrow$

- Independent development
- Scalability
- Flexibility to choose the most suitable technology and frameworks
- Can update app frontend independently of backend
- Reuse backend services across clients (mobile, Web, etc.)

2.5.2 Backend/Network Services

- **Authentication and authorization**
 - user sign-up, sign-in, and sign-out processes; access control and permissions
- **Data storage and mgmt**
 - user data and app content, typically with a CRUD interface
- **Real-time data sync**
 - real-time updates of data across user devices
- **Payment processing**
 - in-app purchases, subscriptions, payment verification and receipts
- Content Delivery Networks (CDNs)
- Analytics and reporting
- Email and SMS
- Backup and recovery

- Geo-location
- Integration services
- ...

2.5.3 Push Notifications

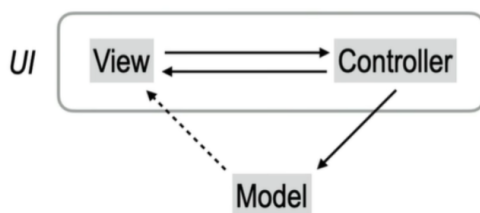
- Messages sent to the user by the backend of the app
- Each platform has its own push notification service

Push notification flow

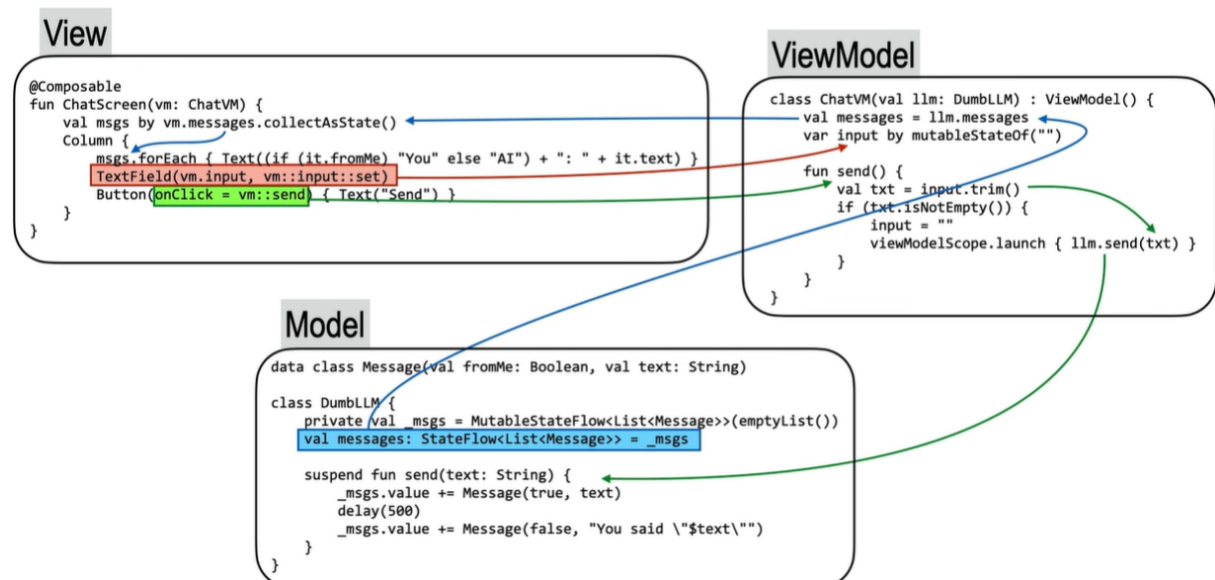
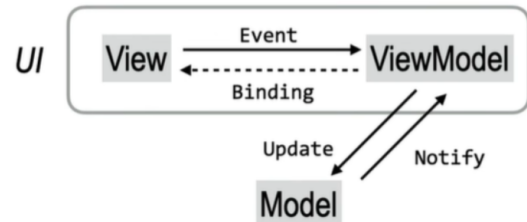
- **Push service registration:** the device requests a token from the platform's push service (Google/Apple), receives it, and registers that token with the app's backend (notification servers).
- **Routine delivery process:** the backend sends the notification to its push servers, which forward the message to the platform's push service, which delivers the push message to the device.

2.6 MVVM

MVC (Model View Controller)



MVVM (Model View ViewModel)



Observability

Key concept in MVVM: enables communication between the app's data layer and its UI.

- The view never pulls data directly from the view model; it reacts to changes pushed from the view model through the underlying mechanism.
- The view model exposes state; the UI functions in the view connect to this state; once bound, the framework recomposes the affected UI functions automatically whenever there's a state change.
- When the user interacts with the UI, the view calls the view model to communicate these user actions.

Remark

Layers are decoupled $\Rightarrow$ you can swap the model (e.g. dumb echo LLM for one calling OpenAI or a local model) or the view (e.g. phone UI for a web browser) without touching the other layers.

Where does “3 minutes ago” belong?

To show in the UI how long ago a message was sent (e.g. “3 minutes ago”), which layer holds that functionality: the view, the view model, or the model?

Answer: the view model

- The model keeps the timestamp of each message – set once, it never changes.
- The string on screen changes every minute while the data sits still, so it can't be done in the model.
- Not in the view either: the moment the view does data arithmetic and runs a timer, you can't test any of it without launching a screen – the view must stay stateless (re-creatable from the view model).
- The view model exists exactly for this kind of UI logic: to display “3 minutes ago” you need the model's timestamp plus a notion of the current time, both combined in the view model – testable in milliseconds by feeding it various values of current time.

Variant: for a screen saying “last refreshed 2 minutes ago”, the timestamp of the last refresh belongs in the view model instead of the model, because nobody sent that information and no other client cares – purely UI state, nothing in the model.

2.6.1 Benefits of MVVM

- Decouple UI logic from application's logic (“business logic”)
 - View is conceptually a function of the ViewModel $\Rightarrow$ can be entirely computed from the ViewModel every time
 - E.g., on mobile: app may run in background and need no UI — can destroy View and re-create from the ViewModel whenever needed
- Enables UIs that are reactive, responsive, dynamic
- Easy to test
- Easy to develop by separate people
- Easy to evolve, maintain, and reuse

2.7 Components of A Mobile App

2.7.1 Four Android App Components

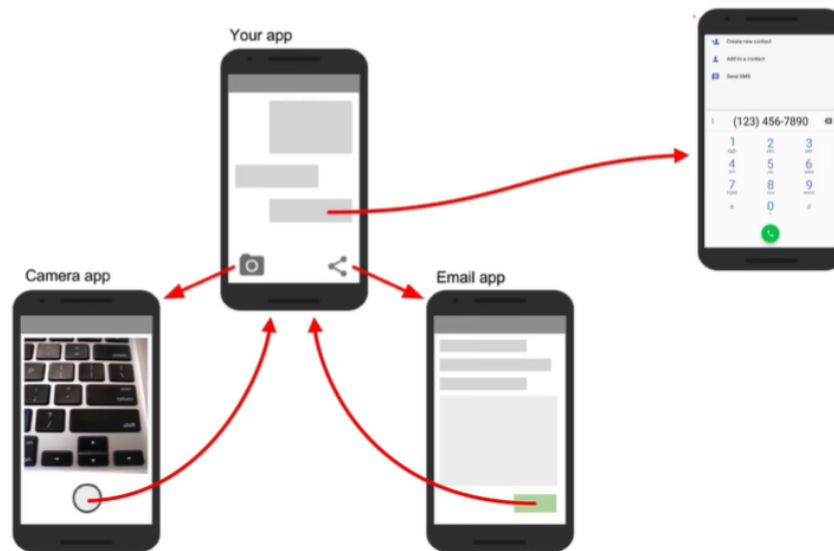
- [1] **Activities**
 - Entry point for interacting with the user (1 screen + UI)
 - One app can start an activity in another app (e.g., camera can email photo)
- [2] **Services**
 - Run in background without UI
 - (often replaced or augmented by WorkManager for deferrable tasks)
- [3] **Broadcast receivers**
 - Receive “out of band” event broadcasts
 - App doesn’t need to be running to receive broadcast
- [4] **Content providers**
 - Manage data that is of interest to multiple apps

2.7.2 Activities

- Key part of an Android app
- Provides the window in which the app draws its UI
- UI consists of multiple screens
- Screens defined inside the main Activity

2.7.3 Inter-App Communication via Intents

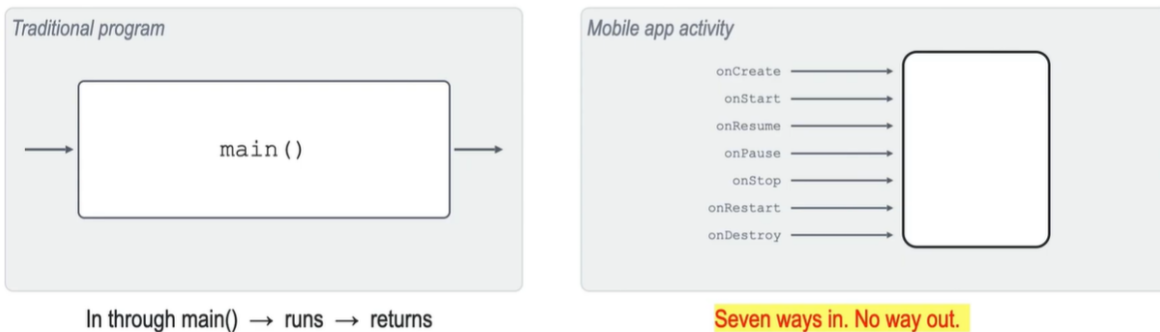
- An app can cause another app’s component to start
 - E.g., WhatsApp takes a photo with the Camera app
- Activation is via an Intent object
 - Think of it as an “active message”
- To start activities and services: `<action, URI>` → explicit intent
 - E.g., `<ACTION_DIAL, "tel:123456789">`, `<ACTION_WEB_SEARCH, "app developers">`
- Register for broadcast announcements: `<announcement>` → implicit intent
 - E.g., `<ACTION_BATTERY_LOW>`, `<ACTION_POWER_DISCONNECTED>`
- For content providers: pull instead of push
 - Activated by request from a `ContentResolver`

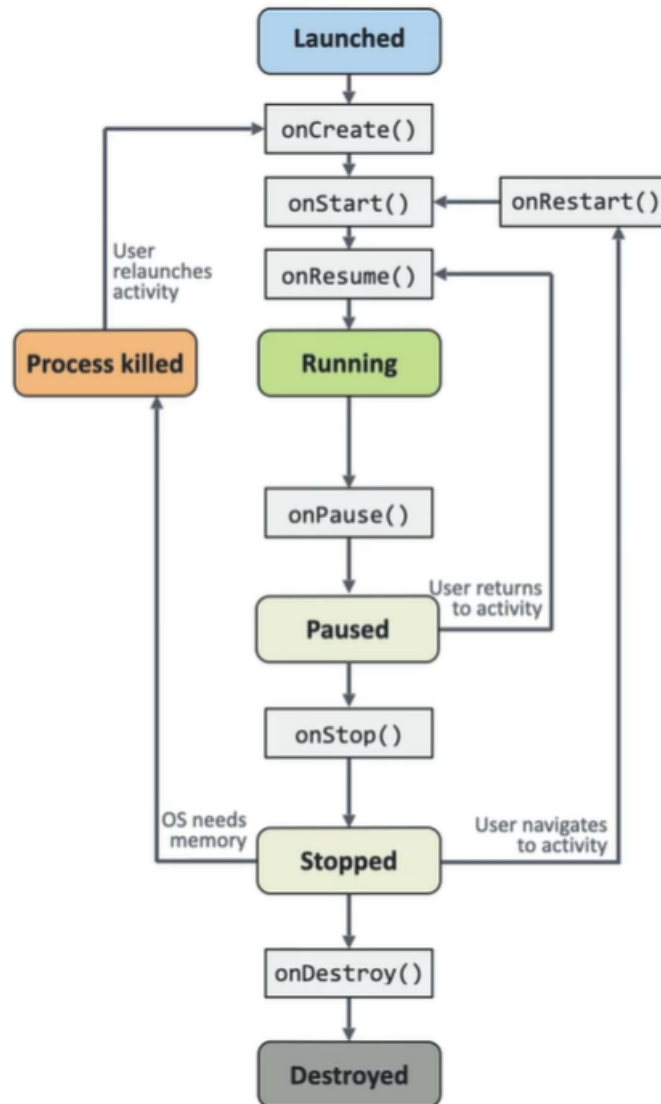


2.8 The Lifecycle of an Android Activity

Single entry vs. multiple entry points

A traditional program has one entry point, `main(): in → runs → returns`. An Android Activity instead exposes **seven** lifecycle callbacks, each a possible entry point – and no single way out.





Rotating mid-form

The user is halfway through filling in a form – three fields done, typing the fourth – then rotates the phone to landscape. Which callbacks fire from the start of the rotation until the form is back on screen, and what happens to the text typed so far?

Answer

The Activity object is destroyed completely, then a brand new one is created in landscape mode:

pause → stop → destroy → create → start → resume

with a **different Activity object** and a different screen configuration – anything kept in the Activity's own fields is gone.

Why the form text survives anyway

- The **ViewModel** is deliberately scoped to outlive the Activity across a configuration change. If the form text lives in the ViewModel (as it should), the new Activity reads that state straight back – separation of concerns at work.
- Redrawing itself is not a callback: a frame loop repaints the screen at 60–120 Hz whenever something changed. The lifecycle governs whether the UI is drawable; the frame loop governs when pixels actually change.

A different risk: switching away

Switching away from the app mid-form (not a rotation – e.g. to take a call) risks the process itself being killed before it resumes. That kills everything, including the ViewModel. The only way to recover the text would be to persist it in **onStop** – but a half-filled form generally isn't expected to survive this case.

iOS has a similar lifecycle

Direct correspondences exist between the Android and iOS callbacks. Difference: iOS may enter a **suspended** state after being paused (stays in memory, executes no code); Android may keep the process alive with no explicit suspended state.

Key Takeaway

Mobile app lifecycles are considerably more sophisticated than server or desktop ones – a reflection of the added complexity and constraints mobile apps face.

CHAPTER 3

Week2 : Part A

Validation vs. Verification

- **Validation** asks whether the requirements themselves are the right ones.
- **Verification** checks that the product actually satisfies those requirements.

The goal of testing is to find bugs so they can be fixed before release.

3.1 Testing: Different Levels

(recap from CS-214)

- **Unit tests** exercise individual components or functions in isolation — e.g. a user-input validation function.
- **Integration tests** exercise combinations of components together — e.g. the login interface talking to the backend database.
- **System tests** exercise the complete, integrated software — e.g. the entire app's functionality.
- **End-to-end tests** follow a full user journey from start to finish. Every e2e test is a system test, but not every system test is end-to-end.
- **Acceptance tests** exercise the product in real-world scenarios to confirm it meets stakeholders' requirements.

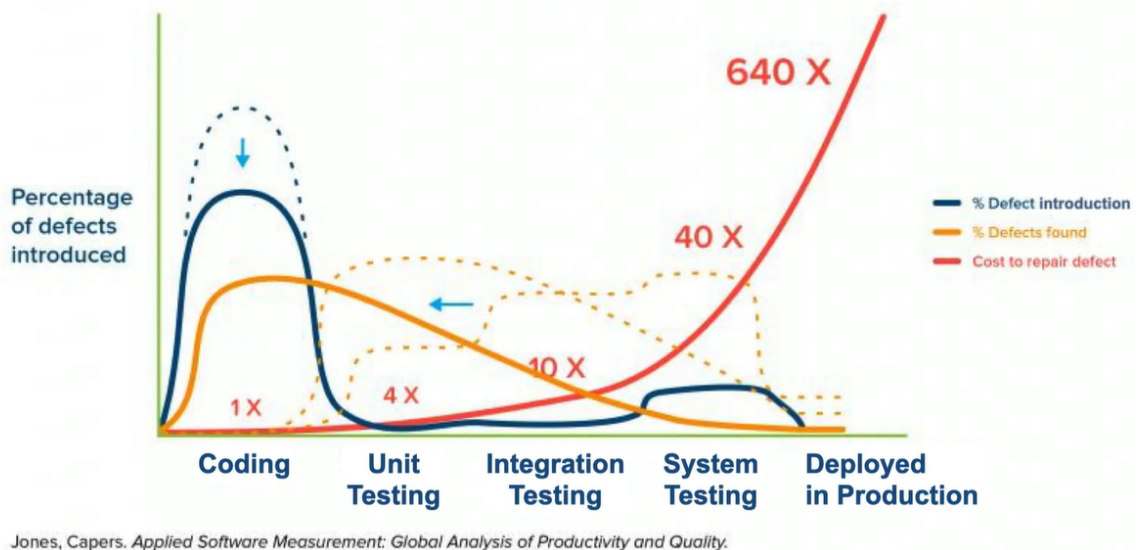
3.1.1 Writing Good Unit Tests

- **Keep tests small, checking only one thing**
- **Keep tests independent of each other**
- **Decouple tests from the rest of the implementation**, as much as possible, especially around object creation

3.2 The Cost of Bugs

More causes of increasing bug costs:

- **Technical debt**
 - bug workarounds $\Rightarrow$ codebase becomes more complex and fragile
- **Lost knowledge**
 - the longer a bug lives, the more likely the original developers have moved on
- **Entrenchment** — the bug has become part of the interface
 - other code and other systems come to depend on the buggy behavior, so eventually you can't fix it without breaking them



3.3 How Well Can We Test?

Dijkstra's Law

Program testing can be used to show the presence of bugs, but never to show their absence!

Edsger W. Dijkstra

Notes on Structured Programming (1970)

- Cannot test all behaviors (too many)
- **Goal of testing**
 - identify bugs in order to fix them
 - gain confidence in software quality
 - statistically verify that program meets requirements
- How do we tell how much testing we still need to do?

3.4 Coverage Metrics

- **Statement coverage**
 - easy to compute, easy to reason about
 - a decent first-cut approximation of how good your tests are
- **Branch coverage**
 - more complicated but still doable
 - a better estimate of the quality of your tests
 - basis-set testing (cyclomatic complexity) gives us the sets for 100% branch coverage
- **Path coverage**
 - impractical
 - the perfect measure of how good your tests are

3.5 Technical Debt

- **Implied cost of additional rework caused by doing a hack**
 - fixing hard-coded values, poorly structured code, coding-convention violations, ...
- **Causes**
 - Rushed timelines, incomplete requirements, lack of knowledge, legacy code, evolution
- **Kinds of technical debt**
 - Deliberate, Inadvertent, Bit Rot
- **Managing technical debt**
 - Recognize, Measure, Devise a repayment plan, Prevent

3.6 Code Smells

- **Code patterns that suggest a potential issue or problem**
 - not bugs, just indicators of increased risks for future bugs, maintainability challenges, etc.
- | | |
|---|---|
| <ul style="list-style-type: none"> • Large Class • Long Method • Primitive Obsession • Feature Envy • Duplicated Code • God Object • Data Clumps | <ul style="list-style-type: none"> • Shotgun Surgery • Lazy Class/Freeloader • Speculative Generality • Temporary Field • Message Chains • Middle Man |
|---|---|



Maintainability overview

Each bubble is one file: **lines of code** on the x-axis, **technical debt** (estimated time to fix) on the y-axis, **bubble size** is the number of code smells, and **color** is the maintainability rating (A, healthy, through E, critical).

- Most files cluster near the origin: small, low-debt, rated A — the bulk of a healthy codebase.
- The outliers — large bubbles, or bubbles shifted toward orange/red — are the files worth refactoring first: many smells and/or high estimated debt relative to their size.
- The right-hand chart is the same view at a larger scale (debt in days rather than hours), showing how smells and debt compound as files grow.

3.7 What Coverage Doesn't Catch

3.7.1 The Imperfection of Coverage

- High coverage $\Rightarrow$ well-exercised codebase $\nRightarrow$ freedom from defects
- **Coverage is an imperfect proxy metric for test quality**
 - measures how much code is executed by tests, not how well they test the code
 - does not measure the quality and importance of the test scenarios
- **Challenge: Not all code is equal**
- **Challenge: Flaky tests**
- **Challenge: Contextual sensitivity**
 - the more circumstances need to combine in order for a bug to manifest, the harder it is to anticipate and reveal them in testing (e.g., concurrency issues)

3.7.2 Smartphone Platform: Heterogeneity Challenges to Testing

- **Devices**
 - device-specific or configuration-specific bugs
- **Integration with external systems**
 - you depend on the network!
 - unexpected behavior can cause mobile app to misbehave
- **Users**
 - watch your input validation
- **Sensors are finicky**
- **Interaction with system features**

3.7.3 Testing Challenges: Non-Functional Requirements

- performance, security, usability, compatibility, etc.
- test coverage numbers do not capture these directly
 - yet they can significantly impact user experience
- static analysis tools
- performance and load testing
- security testing (both automated and manual)
- exploratory testing

3.7.4 Careful What You Ask For

An agent optimizes exactly the metric you name

- ask for “increase coverage to 80%” and you get the two tests we opened with
 - ask for tests of specific behaviors, and you get tests
-
- **For every test: what would have to break for this to fail (i.e., “go red”)?**
 - if you can't answer in one sentence, it isn't a test — delete it
 - **Verify the suite, not the number**

- break your own code on purpose; if nothing goes red, your tests are decorative
- **A coverage jump is a reason to look harder, not to celebrate**
 - 40% to 90% in one afternoon could mean 40 plausible green tests nobody has read

3.8 Takeaways

Key Takeaway

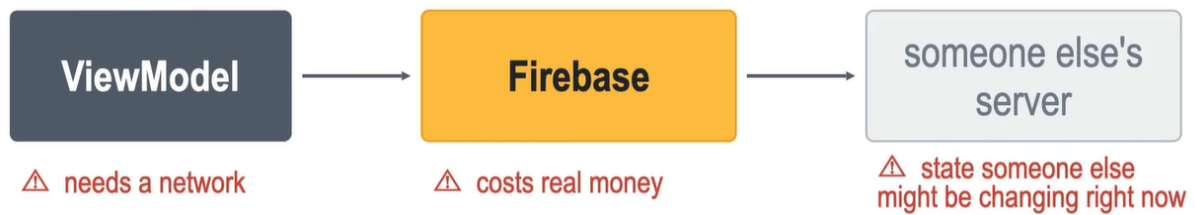
- **Coverage metrics**
 - just an indicator, they do not “sign off” on software quality
 - provide guidance: areas that need attention
- **Tests are code**
 - update tests as code evolves (e.g., every time you fix a bug, write a test for it)
 - keep tests under version control
 - run often
 - should be of production quality
- **Look at the big picture**
 - consider technical debt, code smells, etc.

CHAPTER 4

Week2 : Part B

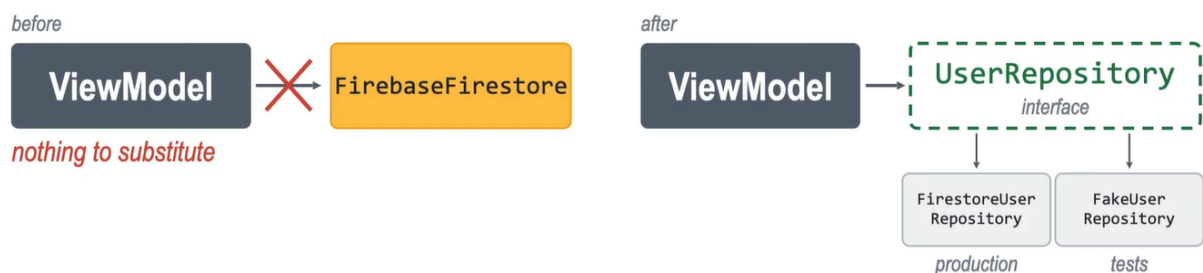
4.1 Test Doubles Mocks, Stubs, Fakes

4.1.1 Testing a ViewModel That Talks to the Network



Slow, flaky, different answers on different days ⇒ not a test.

4.1.2 Breaking the Dependency



Program against an interface, not a concrete class

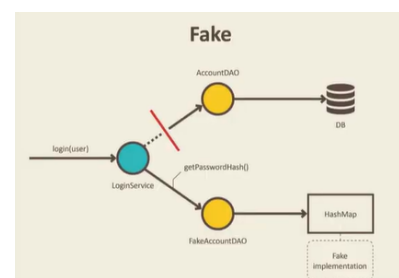
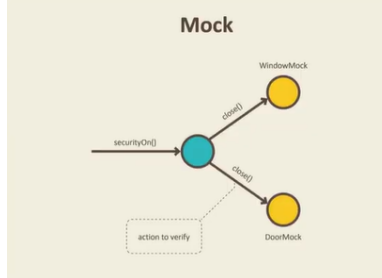
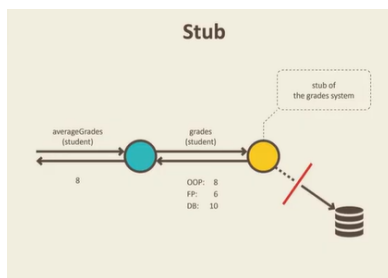
- **Before** — the ViewModel calls `FirestoreFirestore` directly. There is nothing to substitute: production code and test code are the same code.
- **After** — the ViewModel depends only on a `UserRepository` interface. `FirestoreUserRepository` implements it for production; `FakeUserRepository` implements it for tests.

This is what makes stubs, mocks, and fakes possible in the first place: you can only inject a test double where the code depends on an interface, not a hard-coded implementation.

	Fake (<i>in-memory</i>)	Emulator (<i>real Fire- base, local</i>)	Real Firebase (<i>cloud</i>)
<i>speed</i>	milliseconds	seconds	seconds + network
<i>checks</i>	your own logic	your queries and secu- rity rules	nothing you should be unit-testing
<i>use for</i>	unit tests (hundreds of them)	integration tests (a handful)	never in CI

4.1.3 Test Doubles

STUB	MOCK	FAKE
<i>answers questions</i>	<i>answers, and remembers</i>	<i>a small working thing</i>
No logic. A canned value. You never ask it anything afterwards.	Keeps a record of every call: arguments, how many times, ...	A real implementation, simplified. An in-memory map, not a database.
<i>assert on...</i> your own code's result	<i>assert on...</i> calls your code made to the mock	<i>assert on...</i> your code's behavior over a sequence of interactions w/ fake

**Remark**

- **Stub** — `grades(student)` is cut off from the real DB and replaced by a canned answer; nobody checks that the cut edge was ever used.
- **Mock** — `securityOn()` fans out to `WindowMock` and `DoorMock`; the test verifies that `close()` was sent to both.
- **Fake** — `LoginService` is cut off from `AccountDAO` and instead wired to `FakeAccountDAO`, a real (tiny) implementation backed by a `HashMap` instead of the DB.

a

Check that your `ViewModel` shows an error message when sign-in fails.

b

Check that a failed sign-in is reported to analytics exactly once.

c

Test a cart flow that reads and writes twenty user documents.

STUB

You care what your `ViewModel` does with a failure, not how the failure was produced.

MOCK

"exactly once" is a claim about the conversation. Only a recorder can check it.

FAKE

Twenty documents, order matters, later reads depend on earlier writes. Ten lines of in-memory map.

4.2 Dependency Injection

Dependency Injection (DI)

A class does not create the objects it depends on itself; instead, those dependencies are handed to it ("injected") from the outside, typically as constructor parameters. The class only knows about an **interface**; it never decides, and never even sees, which concrete implementation it gets.

Without DI vs. with DI

- **Without DI** — the ViewModel builds its own dependency internally, e.g. `private val repo = FirestoreUserRepository()`. It is hard-wired to that one concrete class: nothing can ever be swapped in, in production or in a test.
- **With DI** — the ViewModel's constructor takes a `UserRepository` parameter instead: `class UserViewModel(private val repo: UserRepository)`. Whoever *creates* the ViewModel decides which implementation to hand it — `FirestoreUserRepository` when the app runs, `FakeUserRepository` when a test runs — the ViewModel's own code never changes.

This is exactly the “after” picture from §4.1.2: DI is simply the practice of always constructing a class this way, so that a test double can be substituted without touching the class under test.

Who does the injecting?

Someone still has to build the real object graph and pass the right implementation to the right constructor. **Manual DI** means you write that wiring code yourself — fine for a handful of classes, as the next slide shows.

Manual DI: pros vs. cons.**GOOD**

- Testability (swap in a fake, get isolated deterministic tests)
- Flexibility (diff implementations for diff scenarios)

BAD

- Scalability (as the app grows you are hand-managing a graph of dependencies)
- Boilerplate (one factory per ViewModel, all of it near-identical)
- Consistency (7 teammates will do this in 7 ways!)
- Lifecycle (easy to leak a Context and never notice)

OK to do by hand for a small project. But not at industrial scale.

Why frameworks exist

Manual DI does not scale: every new ViewModel needs its own hand-written factory, the wiring code is near-identical across them (boilerplate), each teammate wires things up slightly differently (inconsistency), and it is easy to leak a long-lived object like a `Context` without noticing. **DI frameworks** (e.g. Hilt, Dagger) generate that wiring code for you: you declare which implementation satisfies which interface once, and the framework builds and injects the right objects everywhere they are needed.

4.3 Regression Testing**Regression**

A **regression** is a bug that reintroduces a problem that was already fixed once — the software regresses to a previously broken state, usually because a later change did not account for it.

Rule #1

When you fix a bug, add a test that fails on the old code and passes on the fix. That test now proves the bug's absence **forever**: it turns every future change into a check against this specific regression, not just a one-time fix.

Rule #2

A software product is never “done” — it keeps evolving, and **every change risks breaking something that already worked**. This is exactly why Rule #1 matters: each regression test accumulated over time is a fixed cost paid once, guarding against a risk that recurs on every single future change.

4.3.1 Building a Regression Suite*Step 2: identify the core features*

Before writing regression tests, list the product's core features and rank them — not every feature deserves the same testing budget. A **core features matrix** records, per feature, its description, importance, user impact, and any testing notes; the highest-importance, highest-impact rows are exactly where regressions hurt the most and where the suite should concentrate first.

Regression Suite — Step 2: Core Features Matrix

Feature	Description	Importance	User Impact	Notes
User Registration	Allows new users to create an account	High	Essential for user engagement and personalization	Critical for first-time users
Login/Logout	Secure authentication mechanism for users	High	Essential for accessing personalized features	Must support social media logins
Restaurant Search	Enables users to search for restaurants by name, cuisine, or location	High	Core to the app's functionality	Should include filters for dietary restrictions
Menu Browsing	Displays available dishes from selected restaurants	High	Directly impacts ordering process	Include photos and descriptions
Order Customization	Allows users to customize orders (e.g., ingredient modifications)	Medium	Enhances user satisfaction	Critical for users with allergies
Shopping Cart Management	Supports adding, removing, and modifying items in the cart	High	Essential for order processing	Include an option to save the cart
Checkout Process	Processes orders with payment and delivery options	High	Critical for completing transactions	Must be secure and support multiple payment methods
Order Tracking	Enables users to track the status of their orders in real-time	Medium	Improves user experience	Push notifications for status updates
Ratings and Reviews	Allows users to rate and review restaurants and dishes	Low	Useful for community engagement	Filter and flag inappropriate content
User Profile Management	Lets users update personal information, payment methods, and order history	Medium	Important for personalization	Include privacy settings
Push Notifications	Sends notifications for order updates, promotions, and app updates	Medium	Enhances user engagement	User option to customize notification preferences
Help and Support	Provides access to FAQ, contact options, and support tickets	Low	Supports user inquiries and issues	Include chat support if possible

4.3.2 Regression Test Suite: Summary**Key Takeaway**

- **Goal of the regression suite** — spot regressions on both functional and non-functional requirements.
- **The suite needs to run frequently** — which means it needs to be fully automated and fast.
- **Use CI to run it automatically** whenever needed, e.g. on every push.

4.4 UI Testing for Your App

VIEWMODEL TESTS

~5 ms each, no screen

- losing connectivity sets `state.isOffline`
- being offline sets `state.canSubmit` to false
- an already-signed-in user stays signed in

ONE UI TEST

3–10 s, needs an emulator

with `isOffline = true`, the banner really is visible and the button really is disabled

UI tests check the wiring. ViewModel tests check the logic.

Remark

The ViewModel test is cheap and covers many cases; the UI test is expensive but is the only one that catches wiring mistakes — e.g. the banner's visibility bound to the wrong field, even though the ViewModel's own logic is correct.

4.4.1 Where Does Validation Belong?

IN THE VIEW · format

- is a mandatory field empty?
- does this look like an email address?
- does the phone number have the right number of digits?
- are the characters allowed at all?

IN THE VIEWMODEL · business rules

- is this username already taken?
- is that area code one we serve?
- does this order exceed the user's credit?
- is this booking in the past?

4.4.2 What to Test with UI Tests

Definition

- **Navigation** — e.g. pressing this button takes you to that screen.
- **Permission flows** — e.g. the permission system dialog appears, and both answers are handled.
- **End-to-end test of the critical path** — e.g. the one crucial flow that your app exists for.
- **Regression** — e.g. UI wiring bugs.

Have a few such UI tests, not tens.

4.4.3 UI Testing: A Recap

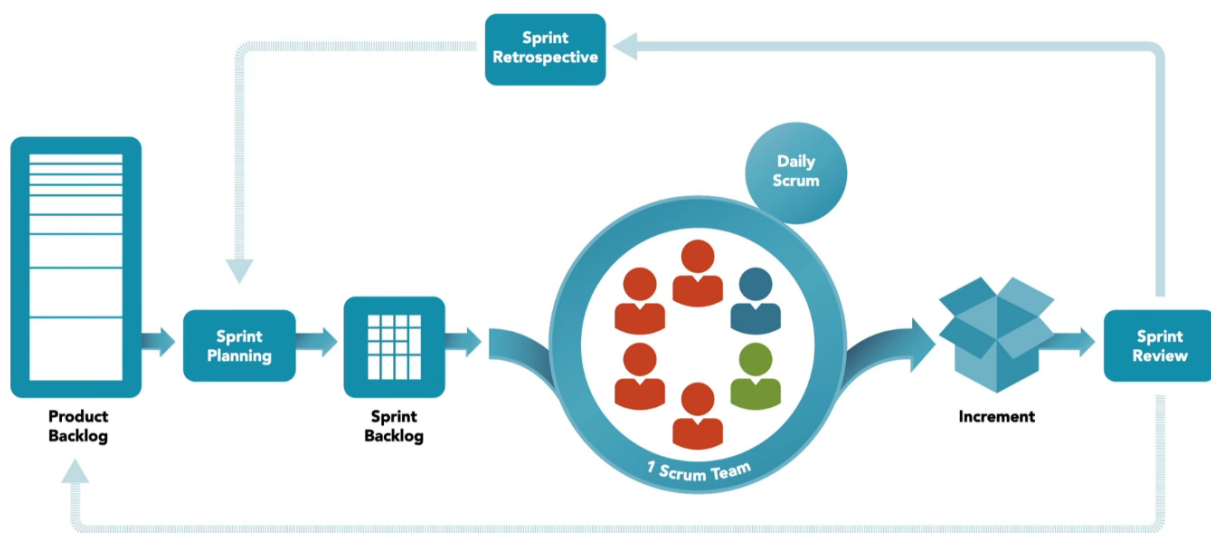
Key Takeaway

- Check that the ViewModel's state reaches the screen correctly.
- Everything else belongs below the UI — *there it can run orders of magnitude faster and isn't flaky.*
- Tag what you test, then assert on state (not on appearance).

CHAPTER 5

Week3 :

5.1 Scrum



Work is organised in **sprints**, short fixed-length iterations. The **Product Backlog** lists everything the product could need, ordered by priority. At **Sprint Planning**, the team picks the items it commits to for this sprint, which form the **Sprint Backlog**. During the sprint, the **Scrum Team** works on those items and syncs every day in a short **Daily Scrum**. The sprint produces a working **Increment** of the product.

Two feedback loops

- **Sprint Review** — the increment is shown to stakeholders; their feedback goes back into the **Product Backlog** (*what we build*).
- **Sprint Retrospective** — the team reflects on how the sprint went; the improvements feed the next **Sprint Planning** (*how we work*).

5.2 Continuous Integration / Continuous Delivery & Deployment

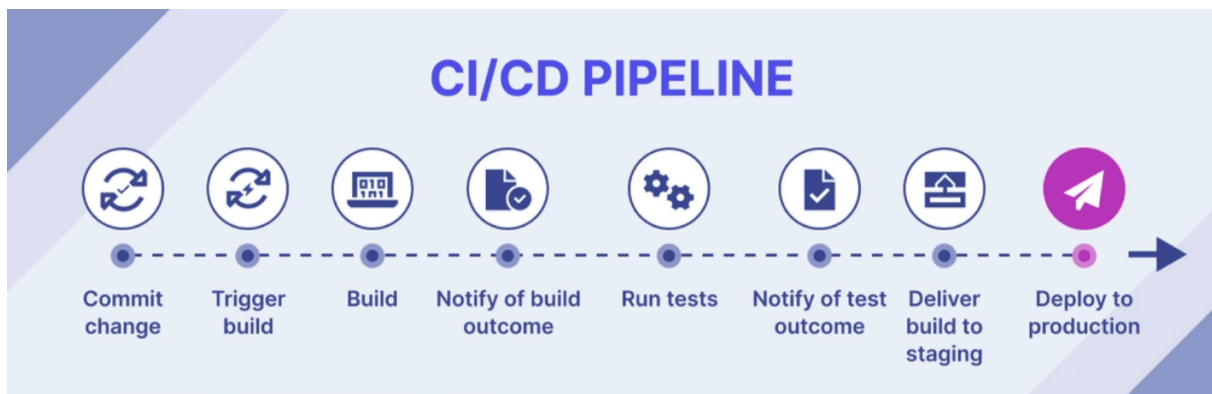
CI/CD

An **automated assembly line** for your software product: every change a developer pushes travels through the same fixed sequence of steps (build, test, package, deploy) without anyone having to run them by hand.

Three stages, three sets of goals

- **Continuous Integration** —
 - merge code changes from multiple devs and ensure that they work together;
 - catch bugs and integration problems as early as possible.
- **Continuous Delivery** —
 - constantly have the software in a “deployable state”;
 - automatically package the software, ready for deployment.
- **Continuous Deployment** —
 - automatically deploy to testing / staging / production.

Each stage builds on the previous one: integration guarantees the merged code works, delivery guarantees it can be shipped at any moment, and deployment actually ships it.

5.2.1 The CI/CD Pipeline

A single commit is enough to set the whole line in motion: the build is triggered automatically, then the code is compiled and tested, and the team is **notified after each stage**, so a failure is spotted minutes after the push. The first steps (commit → test outcome) are **CI**. Delivering the build to staging is **Continuous Delivery**. Pushing it to production automatically is **Continuous Deployment**.

Delivery vs. Deployment

The difference is who presses the final button. With **Continuous Delivery**, every change ends up as a release-ready package, but a human decides when to ship it. With **Continuous Deployment**, that last step is automated too: any change that passes the pipeline goes straight out to testing, staging or production.

5.2.2 Our Pipeline for the Bootcamp



What it buys you

Your reviewer never has to comment on a brace. They get to spend all their attention on whether the change is right.

What it does not buy you

Green is not a claim about correctness. It simply means that some number of tests succeed.

A concrete, smaller version of the pipeline above: every push to GitHub is built, then automatically **formatted and linted**, then tested on an **Android emulator**. It stops at CI (there is no delivery or deployment step). Because formatting and style are checked by a machine, code review can focus on what matters. But a green check only says that the tests we wrote pass; it can never be better than the test suite behind it.