The *Master* Build Guide

The operational adapter layer. Model selection matrix, nine-prompt library, twelve AI failure modes, daily/weekly rituals. When models change, only this document updates.

MODEL SELECTION MATRIX · APRIL 2026

Right model.
Right task.

Task types are stable. The "Current Pick" column is the only thing that changes when a new model releases.

TASK TYPE	MODEL CLASS	CURRENT PICK
Architecture decisions, SSOT drafting, spec review	Highest-reasoning	`Opus 4.7`
Bulk code generation (>500 lines/file)	Highest-output-window	`Sonnet 4.6 Extended`
Scientific / domain-accurate content	Highest-reasoning	`Opus 4.7`
Repetitive module/chapter builds		

	Balanced	`Sonnet 4.6`
Quick fixes, small refactors, one-file edits	Fast / cheap	`Sonnet 4.6`
Governance review (second opinion)	Different family	`ChatGPT / Gemini`
SSOT review · Drift reconciliation sweeps	Highest-reasoning	`Opus 4.7`

CODE LAYER · ANTI-DRIFT BY DESIGN

Design Pattern Catalogue. *Structural resistance* to drift.

AI-generated code defaults to the simplest implementation — often the one most vulnerable to drift. These eight patterns, drawn from the MiVA build, produce code that is structurally resistant to drift by design. Each maps directly to a specific failure mode.

P1

Sealed Interface Pattern

Prevents: FM-10 · CD1 · Contract mutation

Core pattern ▾

What it is

A public contract (API, schema, function signatures) is declared immutable for the lifetime of a major version. The contract is implemented separately from the declaration — callers bind to the contract, never to the implementation. This means the implementation can change freely without breaking callers, and the contract cannot drift without a deliberate, logged, reviewed version bump.

> MiVA instance
>
> SealedApi.kt declared 19 functions — the public contract for all Android feature modules. No feature module imports anything outside SealedApi. When CD1 revealed 23 functions in code vs 19 in spec, the pattern itself was the diagnostic — the count mismatch was immediately measurable.

Implementation

```
// SEALED — do not modify
interface SealedApi {
  fun getChapters():
Flow<List<Chapter>>
  fun getClips(chapterId: String):
Flow<List<Clip>>
  fun trackProgress(clipId: String,
pct: Float)
  // ... 16 more sealed functions
}

// MUTABLE — implementation details
class MivaApi : SealedApi {
  override fun getChapters() = ...
}

// Extension — new capability,
sealed intact
fun
SealedApi.getChaptersWithProgress()
=
  getChapters().map {
addProgress(it) }
```

P2

Command Bus / Event Bus Pattern

Prevents: V2 race condition · mutable state drift

Concurrency

What it is

Instead of multiple coroutines writing to shared mutable state, all mutations are expressed as typed commands sent to a single channel. The channel processes commands sequentially, eliminating race conditions by construction. The bus itself is the sealed contract — callers can only send commands, never modify state directly.

MiVA instance — V2 fix

PrefetchEventBus replaced direct mutation of MivaPrefetchManager state. Race condition between multiple coroutines writing to the same prefetch queue eliminated by routing all writes through a single channel.

Implementation

```
sealed class PrefetchCommand {
  data class Start(val chapterId: String) :
PrefetchCommand()
  data class Cancel(val chapterId: String) :
PrefetchCommand()
  object ClearAll : PrefetchCommand()
}

// Single channel — sequential processing
val PrefetchEventBus = Channel<PrefetchCommand>
(BUFFERED)

// No direct state mutation allowed
suspend fun startPrefetch(id: String) {
  PrefetchEventBus.send(PrefetchCommand.Start(id))
}
```

P3

Smart Retry with Error Classification

Prevents: V5 · O4 RestException gap · silent failures

Resilience

What it is

AI-generated retry logic typically catches `Exception` broadly — which swallows `CancellationException` and retries on

Implementation

```
sealed class RetryDecision {
  object Retry : RetryDecision()
  object NoRetry : RetryDecision()
```

errors that should fail immediately (auth failures, data errors). Smart retry classifies errors first: transient errors get retried with exponential backoff, permanent errors fail fast, cancellations propagate immediately.

> Open item O4 in MiVA
>
> RetryPolicy.shouldRetry() only handles IOException/SocketTimeoutException. Supabase SDK throws RestException — 502 gateway errors get NO_RETRY, causing instant user-visible failures on transient outages. Pattern shows how to classify SDK-specific exceptions properly.

```
}

fun shouldRetry(e: Throwable) =
when {
  e is CancellationException ->
throw e // never retry
  e is IOException -> Retry
  e is SocketTimeoutException ->
Retry
  e is RestException &&
e.statusCode in 500..599 -> Retry
  e is RestException &&
e.statusCode == 401 -> NoRetry
  else -> NoRetry
}
```

P4

Atomic Write Pattern

Prevents: V9 · partial writes · corrupted downloads

Data integrity

What it is

AI-generated file write code typically writes directly to the target path. If the process is interrupted mid-write (crash, low memory, network loss), the file is left in a partially-written, corrupted state. The atomic write pattern writes to a temporary file first, verifies integrity with fsync, then atomically renames to the target. The target either contains the complete new content or the previous complete content — never a partial state.

> MiVA V9 fix
>
> AtomicDownloader replaced direct file writes in the video download pipeline. Previously, a crashed download left corrupted .mp4 files that passed extension checks but failed playback — a silent failure that only surfaced at runtime.

Implementation

```
object AtomicDownloader {
  suspend fun write(target: File,
data: ByteArray) {
    // 1. Write to temp file
    val tmp = File("${"
{"}target.path{"}"}.tmp")
    tmp.outputStream().use { out ->
      out.write(data)
      out.fd.sync() // fsync —
flush to disk
    }
    // 2. Verify integrity
    check(tmp.length() ==
data.size.toLong())
    // 3. Atomic rename — never
partial
    tmp.renameTo(target)
  }
}
```

P5

Repository Pattern with Schema Isolation

Prevents: FM-01 · FM-06 · hallucinated column names

DB isolation

What it is

All database access is abstracted behind a Repository interface. The repository is the only layer that knows column names, table structures, and RPC function signatures. Feature code never references column names directly — it calls repository methods. When AI generates a query with a hallucinated column name, the error surfaces at the repository boundary rather than silently returning null data at runtime.

> MiVA enforcement rule
>
> Before writing any function touching DB, run `control.show_columns('schema','table')`. Never assume column names. This is enforced in the SSOT Developer Rules — a process rule made necessary by FM-06.

Implementation

```
// Repository interface — only schema knowledge lives here
interface ChapterRepository {
  suspend fun getChapters(packSlug: String): Result<List<Chapter>>
  suspend fun getClips(chapterId: String): Result<List<Clip>>
}

// Feature code — no column names, no SQL
class PlayerViewModel(private val repo: ChapterRepository) {
  fun loadChapter(id: String) {
    repo.getClips(id) // hallucinated columns fail here
  }
}
```

P6

Structured Concurrency + CancellationException Guard

Prevents: V7 · 44 guard instances added in MiVA v41

Coroutines

What it is

AI-generated coroutine code almost universally catches `Exception` broadly — which silently swallows `CancellationException`. In Kotlin coroutines, swallowing cancellation breaks structured concurrency: parent scopes don't

Implementation

```
// WRONG — AI default — breaks structured concurrency
try {
  val result = apiCall()
} catch (e: Exception) {
```

know the child was cancelled, leading to resource leaks, zombie coroutines, and unpredictable state after navigation events. Every catch block must re-throw `CancellationException` immediately.

> MiVA scale
>
> 44 CancellationException guards added across MiVA v41 beta6. The vulnerability affected 27 MivaApi functions + RetryPolicy + GlobalErrorHandler — every single coroutine-based API call in the app.

```
  Result.failure(e) // swallows
CancellationException!
}

// CORRECT — V7 pattern
try {
  val result = apiCall()
} catch (e: CancellationException)
{
  throw e // NEVER swallow —
propagate immediately
} catch (e: Exception) {
  Result.failure(e) // safe to
handle other exceptions
}
```

P7

Strategy Pattern for Capability Branching

Prevents: nested conditionals · device-specific drift

Branching logic

What it is

AI-generated device/capability branching accumulates as nested if/when blocks — hard to test, easy for AI to mis-state in future sessions (FM-11 attribution laundering). Strategy pattern extracts each capability variant into its own class implementing a sealed interface. Selection is done once at startup or at the boundary; all subsequent code is polymorphic and tests against the interface, not the concrete strategy.

> MiVA instance
>
> DeviceProfile uses 3-tier strategy (LOW/MEDIUM/HIGH RAM) rather than inline API level checks. G1: low RAM detection, G2: safeBlur() skips API < 31. Each tier is a separate implementation, selected once at startup.

Implementation

```
sealed interface PlaybackStrategy {
  fun canHardwareDecode(codec:
String): Boolean
  fun maxConcurrentPlayers(): Int
  fun blurEnabled(): Boolean
}

class LowMemStrategy :
PlaybackStrategy {
  override fun
canHardwareDecode(codec: String) =
false
  override fun
maxConcurrentPlayers() = 1
  override fun blurEnabled() =
false
}

// Selected once — used everywhere
polymorphically
```

```
val strategy =
DeviceProfile.selectStrategy()
```

P8

Optimistic Grant + Background Verification

Prevents: F4 webhook delay · payment UX cliff

Payments / UX

What it is

Webhook-based payment confirmation always has latency — Razorpay webhooks can arrive 10–30 seconds after user payment. AI-generated payment flows that wait synchronously for webhook confirmation create an unacceptable UX cliff where the user sees a spinner after paying. Optimistic grant: immediately grant access on payment success signal from SDK, then verify asynchronously in the background. If verification fails, revoke gracefully with a 1-hour grace period.

MiVA F4 implementation

PaymentManager.kt: optimistic grant on Razorpay SDK callback + background webhook verification. 1-hour grace period prevents false revocation during webhook delays. Subscription status is reconciled with DB on next app launch.

Implementation

```
suspend fun onPaymentSuccess(orderId: String) {
  // 1. Optimistic grant — immediate UX
  subscriptionCache.grant(userId, expiresIn =
1.hour)
  navigator.goToContent()

  // 2. Background verification — async
  scope.launch {
    val verified = webhookVerifier.await(orderId,
timeout = 30.seconds)
    if (verified) {
      subscriptionCache.confirmFromServer(userId)
    } else {
      subscriptionCache.revokeWithGrace(userId,
grace = 1.hour)
    }
  }
}
```

QUICK REFERENCE

Pattern → *Failure Mode* mapping.

PATTERN	FAILURE MODE PREVENTED	MIVA EVIDENCE	ENTERPRISE RISK IF MISSING
P1 Sealed Interface	FM-10 · CD1	19 vs 23 function count drift	Breaking API changes ship silently
P2 Command Bus	V2 race condition	PrefetchEventBus eliminates mutable state race	Data corruption under load
P3 Smart Retry	V5 · O4 RestException	502s getting NO_RETRY → instant user error	Transient outages appear as hard failures
P4 Atomic Write	V9 partial write	Corrupted video files on interrupted download	Silent data corruption in file-based systems
P5 Repository	FM-01 · FM-06	show_columns() rule — hallucinated columns fail at repo boundary	Silent null returns from hallucinated queries
P6 CE Guard	V7 · 44 guards	Every MivaApi function had swallowed cancellation	Resource leaks · zombie coroutines in prod
P7 Strategy	FM-11 attribution	DeviceProfile 3-tier vs inline API level checks	Capability branching becomes untestable
P8 Optimistic Grant	F4 webhook delay	Razorpay webhook latency eliminated from UX path	Payment cliff → user abandonment

PROMPT LIBRARY · CLICK TO EXPAND

Nine prompts.
Reusable infrastructure.

A prompt is not chat text — it is reusable infrastructure encoding hard-won patterns. Every prompt has a phase, a model class, and a purpose.

PL-01 FRAME — Problem Distillation

Phase: FRAME · Model: Highest-reasoning

Forces articulation of problem, core bet, and non-goals before any solution talk.

```
I am starting a new unit of work: {name}.
Before proposing any solution, act as a Socratic partner.
Ask me one question at a time to help me articulate:
  1. The problem (plain language, no jargon)
  2. Who the user is and what constraints they operate under
  3. The core bet — the testable thesis this work assumes
  4. Explicit non-goals (what this is NOT)
  5. A single success metric (one number, not a list)
Stop and wait for my answer after each question.
Do not propose architectures, features, or code.
```

PL-02 SEAL — Interface Contract Drafter

Phase: SEAL · Model: Highest-reasoning

Identifies what MUST be sealed based on known-unstable areas, not speculation.

```
Attached: Problem Statement and any prior code / schema.
Your task: propose the MINIMUM sealed interface manifest.
Rules:
  - Seal only what you already know you'll regret changing
  - For each sealed element, cite WHY
  - Leave everything else explicitly mutable
  - Flag anything where you're unsure — do not seal on speculation
```

PL-03 SPEC — SSOT Surgical Edit

Phase: SPEC · Model: Highest-reasoning

Prevents CD2 (regeneration loss). Never regenerate — always edit surgically.

```
Attached: current SSOT v{N}.
I need to make the following changes: {list}.
Do NOT regenerate the SSOT.
Do NOT rewrite sections not touched by these changes.
Instead:
  1. Quote the exact current text for each change
  2. Propose the exact replacement text
  3. List section and line range affected
  4. Flag any downstream sections needing sync
```

PL-04

BUILD — One-Chat-Per-Unit Opener

Phase: BUILD · Model: Varies by task

The first message in every BUILD chat. Establishes context, references, and the contract.

```
This chat is dedicated to building {unit name}.
Attached (use as authoritative sources, not your memory):
  1. SSOT (the slice relevant to this unit)
  2. Master Build Guide
  3. Last completed sibling as structural reference
  4. Source material (PDF, spec, existing code)
Rules:
  - Before claiming anything about an attached file, view it
  - Do not describe code; quote it
  - When complete, say "Ready for external review"
```

PL-05

BUILD — Source-First Enforcement

Phase: BUILD · Model: Any

Use inline whenever the AI is about to make a claim about existing code. Prevents CD3.

```
Before answering:
  1. View the file you're about to reference
  2. Quote the exact function / line / config you're relying on
  3. THEN propose the change
If the file isn't attached, stop and ask for it.
Do not describe code from memory.
```

PL-06

BUILD — Reality Check Gate

Phase: BUILD · Model: Any

Lightweight pre-commit check. Run before accepting any AI output.

```
Before I accept this output, verify:
  [ ] All file paths referenced actually exist
  [ ] All function / API / column names are real (not invented)
  [ ] The change compiles / is valid against current schema
  [ ] No sealed contract is modified without explicit extension
  [ ] Imports / dependencies referenced are present
If any box cannot be checked: do not apply.
```

PL-07

VERIFY — Second-AI Review Prompt

Phase: VERIFY · Model: Different family preferred

Given to a different AI after "Ready for external review." Bias toward finding problems.

```
You are reviewing work produced by another AI.
Attached:
  1. The SSOT section the work should conform to
  2. The changed files
  3. The Builder's summary of what was done
Find anything the Builder missed, got wrong, or invented.
Output: PASS / FAIL
If FAIL: specific line / file / claim and why.
Bias toward finding problems. A clean pass is suspicious.
```

PL-08

RECONCILE — Drift Sweep

Phase: RECONCILE · Model: Highest-reasoning

Run weekly. Compares five artefact pairs. Open a CD-series incident for any non-trivial drift.

```
Perform a drift sweep across the following artefact pairs:
  Pair A: Code <-> SSOT (counts, signatures, file locations)
  Pair B: Decision Log <-> SSOT (every decision traceable)
  Pair C: DB schema <-> Code queries (every column verified)
  Pair D: Sealed baseline <-> actual files (hash match)
  Pair E: SSOT version <-> Decision Log (version bump discipline)
For each pair: CLEAN / DRIFT DETECTED.
```

```
If drift: exact delta + proposed reconciliation.
Open a new CD-series incident if non-trivial.
```

PL-09

EVOLVE — Extension Request

Phase: EVOLVE · Model: Highest-reasoning

Forces the extension pattern instead of mutation when sealed interfaces need new capability.

```
I need a new capability: {description}.
This appears to require changes to: {sealed element}.
Per the sealed-safe extension rule, do NOT modify the sealed element.
Instead, propose:
  1. An extension that uses the sealed element internally
  2. Backward-compatibility guarantee
  3. Deprecation pathway ONLY if truly necessary
  4. A Decision Log entry capturing the rationale
```

AI FAILURE MODES CATALOG

Twelve failure modes. All *named.* All *countered.*

FM-01 Hallucinated API

AI references get_session_by_tag() — no such function exists. Compiles, fails at runtime.

`PL-05 Source-First · PL-06 Reality Check`

FM-02 Wrong-file Attribution

CD3: fix in MivaReelPlayer.kt. Logic was in ChapterPlayerScreen.kt. Bug persisted.

`PL-05 Source-First Enforcement`

FM-03 Silent Truncation

Chapter 1 HTML cut off mid-JavaScript. No error thrown. Content simply missing.

`Model Matrix §2.1 · length check`

FM-04 Regeneration Loss

CD2: SSOT regenerated, two sections silently dropped. Caught by hash change.

`PL-03 Surgical Edit · Update Policy`

FM-05 Status Drift

CD4: V1–V11 marked "Pending" in SSOT, already fixed in code for weeks.

`PL-08 Drift Sweep · Authority Chain`

FM-06 Confident Hallucination

AI asserts a DB column exists with no verification — stated as fact, not hypothesis.

`PL-05 · control.show_columns()`

FM-07 Premature Solution

AI jumps to code before FRAME is locked — solution before problem is stated.

`PL-01 FRAME Distillation`

FM-08 Context Exhaustion

Long chat starts contradicting earlier outputs — model loses track of prior decisions.

`One-chat-per-unit rule · §5 budget`

FM-09 Sycophantic Agreement

AI agrees with a wrong premise the user stated confidently.

`PL-07 (bias toward finding problems)`

FM-10 Plausible-but-Wrong Structure

Code compiles and looks right but violates sealed contract — caught by SEAL hash.

`Build-hash verification · SealedContractTest`

FM-11 Attribution Laundering

AI cites "the spec" for something that is not in the spec.

`PL-07 · require exact quote + line ref`

FM-12 **Model Capability Drift**

A prompt that worked on Opus 4.6 produces worse output on Opus 4.7 — silent regression.

`§2.3 throwaway experiment rule`

Tool gap | Process gap | AI hallucination | Human oversight

OPERATIONAL CADENCE

Calendar events, not *intentions.*

Daily · 15 min | Weekly · 90 min | Per Milestone | Per Model Release

1. **Sealed contract check** — Did any BUILD chat today touch a sealed contract? If yes, run SealedContractTest before closing.
2. **Git commit** — All changed files committed with meaningful messages? No "WIP" commits at end of day.
3. **Review queue** — Did any unit cross "Ready for external review"? Queue the VERIFY chat for tomorrow.
4. **Decision Log** — Any unknowns flagged today that need Decision Log entries? Log them now.

· MiVA Education