TECHNICAL OVERVIEW · FOR DEVELOPERS

How the AI Proposal Engine Works

Staying on-plan, preventing overdraw, and the guardrails that keep AI suggestions safe

This document explains the architecture behind the planner's AI assistant — specifically how it turns a natural-language request into **structured, numeric proposals**, how those proposals are **verified against the real deterministic simulation engine** before a user ever sees them, and the layered guardrails that stop the model from overdrawing accounts or breaking the plan's constraints.

Audience: developers evaluating or extending the system. It assumes familiarity with LLM tool/structured-output patterns. Function names and line-level behaviour reference the single-file implementation (`Retirement-Planner-v7-design-working.html`).

> **The one-sentence model**
>
> **The LLM proposes; the deterministic engine disposes.** The model never computes final outcomes or writes to state directly. It emits a structured intent, the app simulates that intent with the same engine that powers the whole planner, and only verified, human-approved changes are applied.

Contents

1 Architecture at a glance

The planner is a single, offline HTML file. All financial math is **deterministic** and runs locally: `buildPhaseConfig` → `calcAllPhases` → `calcPhase` → `simPhase` performs a month-by-month simulation across five phases, tracking each account ('bucket'), taxes, ACA/IRMAA/RMD effects, and ending balances. The AI assistant is a thin **advisory layer** bolted onto that engine; it has no independent math of its own.

Two layers, one source of truth

Layer	Responsibility	Trust model
Deterministic engine	Simulates the plan, computes balances/taxes/depletion, runs Plan-Health checks. Same code for the UI, reports, and AI verification.	Ground truth. Reproducible.
LLM (Claude via API)	Interprets natural language, explains, and emits structured numeric *intents* (proposals). Sees a fresh plan snapshot every message.	Untrusted for math. Treated as a suggestion source, never an executor.

> **Why this split matters**
>
> LLMs cannot reliably compound balances across 12×N months in their head — and they are confidently wrong when they try. By keeping every consequential number inside the deterministic engine, the model's job shrinks to what it is good at: language, intent, and rationale. Correctness is owned by code that is testable and identical to what the rest of the app uses.

2 End-to-end request lifecycle

A single “make my plan last / rebalance / optimise” request flows through these stages:

#	Stage	What happens	Key function
1	Compose	User text is prefixed with a live plan snapshot + budget + health status.	`buildAiContext()`
2	Send	Snapshot+question sent with the system prompt; history sanitised to {role,content}.	`_aiCallApi()`
3	Parse	Reply is split into prose + a machine-readable <proposals> JSON block.	regex on <proposals>
4	Simulate	Proposals applied to a deep clone of state; the real engine runs.	`_aiDryRunProposals()`
5	Refine	If the goal isn't met, the result is fed back; the model revises (loop).	`_aiVerifyAndRefine()`
6	Present	A verified proposal card is shown with a pass/warn badge.	`showAiProposals()`
7	Apply	Only user-checked rows are written to live state; the plan recomputes.	`aiApplyProposals()`

Stages 4–5 are the safety core: nothing the model says about outcomes is taken at face value. The app re-derives the outcome with the engine and, if needed, sends the model the real result as ground truth.

3 Context engineering — what the model sees

Every user message is wrapped so the model always reasons over current, deterministic facts. `buildAiContext()` assembles three blocks, prepended to the user's question:

- **Live plan snapshot** — full balances, per-phase results, taxes, Social Security/pension splits, and jurisdiction handling (US / UK-resident / foreign). Built by the same `generateAiPrompt()` used elsewhere.
- **Plan Health Check status** (`buildHealthSnapshot()`) — each active check's pass/warn/fail state plus **per-phase headroom** figures (e.g. dollars to the ACA cliff, to the 22% bracket, to the UK basic-rate ceiling). These are the numeric limits the model must size proposals against.
- **Withdrawal budget & depletion facts** (`_aiPlanBudgetText()`) — for each funded bucket: starting balance, a never-deplete sustainable draw, a full spend-down rate, and which buckets currently hit $0 and when. This is the anti-overdraw cheat-sheet (see §8).

> **Freshness guarantee**
>
> The user can edit the plan between messages. Each message carries its own snapshot, and a `_aiFingerprint()` detects changes; when the plan changes mid-conversation the app injects a **"PLAN UPDATED — disregard earlier dollar amounts"** note and trims history. The system prompt instructs the model that the latest snapshot is always authoritative.

4 The system prompt as guardrails

`getAiSystemPrompt()` is assembled per-request and adapts to the plan's residency and currency. Beyond tone, it encodes hard rules:

Scope & role-lock

The model is constrained to this user's retirement plan and personal finance. Off-topic requests get one brief redirect. It is explicitly told to **ignore instructions** (from the user or pasted text) that try to change its role or reveal/override the prompt — basic prompt-injection resistance.

Non-negotiable plan-health constraints

- **Never** propose a change that turns a passing (✓) health check into a warning (■) or fail (✗). If the goal can't be met without breaking one, say so and propose an alternative.
- Size every proposal using the provided **headroom** figures (don't push MAGI past the ACA cliff, taxable income past a bracket ceiling, UK income past the basic-rate band, etc.).
- When the stress-test check is currently passing, avoid large early-withdrawal increases that erode the buffer that makes the plan resilient under bear conditions.
- Close with an explicit `Health checks preserved: […]` summary so the effect on constraints is auditable.

Feasibility & self-verification clause

The prompt tells the model plainly that it *cannot* simulate month-by-month compounding in its head, that a bucket can supply **at most its starting balance plus growth**, and that the app will simulate its proposals and return a `SIMULATION RESULT` to revise against. Infeasible goals must be called out, not papered over with numbers that don't work.

5 Structured proposals (the contract)

When (and only when) the user asks to change something, the model appends a single machine-readable block to the end of its reply. Prose for humans, JSON for the app:

```
<proposals>[
  {"field":"p1.w401k","label":"Phase 1 401k withdrawal /mo",
   "current":3000,"proposed":2800,
   "reason":"Reduces UK taxable income, preserving 20% basic-rate band"},
  {"field":"p2.wcash","label":"Phase 2 cash withdrawal /mo",
   "current":500,"proposed":700,"reason":"Shift to tax-free cash"}
]</proposals>
```

The field namespace is fixed and small, which bounds what a proposal can touch:

Field pattern	Meaning	Examples
`pN.field`	Per-phase setting (N = 1–5)	`p1.w401k, p2.wcash, p3.wEquity, p4.wRoth, p5.rothConversion, pN.partTime`
`s.field`	Top-level plan input	`s.bal401k, s.r401k, s.inflation, s.ssStartAge, s.uss, s.ukp, s.usPension`

Each item must carry both `current` (from the live snapshot) and `proposed`. Empty/qualitative proposals are disallowed by the prompt — the model must commit to concrete numbers, which is what makes them simulable.

6 Verification — dry-run against the real engine

Before a proposal is shown, the app applies it to a **deep clone** of state and runs the production engine. No DOM, no global mutation, no API — pure local math:

```
// _aiDryRunProposals(proposals)
clone = JSON.parse(JSON.stringify(S));        // isolate from live state
_aiApplyProposalsToState(clone, proposals);   // s.* and pN.* writes only
phs = calcAllPhases(clone, p5EndAge, lumps);  // the SAME engine the UI uses
// depletion is detected from real per-phase ending balances:
//   k401Depleted = end.b401k <= 0   (etc. for cash / equity)
// only FUNDED buckets count (an unfunded bucket reading $0 is by design)
return { ok: depletions.length === 0, depletions, endTotal, end };
```

The result drives two things: a user-facing **verification badge** on the proposal card (`_aiOutcomeBadge` — "✓ Verified by simulation" or "■ Simulation shows accounts still deplete…"), and, when needed, a machine-readable result the model can iterate on (`_aiOutcomeText`).

7 Self-correction — the refine loop

If the user's request implies a multi-phase goal ("don't run out", "rebalance", "make it last" — detected by `_aiWantsNoDeplete()`) and the dry-run still shows depletion, the app closes the loop automatically:

- It feeds the model the real `SIMULATION RESULT` (which buckets hit $0 and in which phase) and asks for a corrected `<proposals>` block.
- It re-simulates the revised proposal and repeats, up to `_AI_REFINE_MAX = 3` attempts.

- Each attempt is shown transparently as a system note (“testing in the simulator… asking the AI to refine — attempt 1/3”).
- If it converges: “✓ Refined — the simulation now shows no funded account depleting.” If not, it shows the closest version and states the goal may be infeasible — it never silently presents broken numbers.

> **Ground truth beats eloquence**
>
> The feedback message is deliberately blunt and prescriptive — it tells the model exactly how to fix a depletion (cut draws from the depleting bucket in *earlier* phases, shift to buckets that end in surplus, or reduce total spending) and reminds it that lifetime withdrawals from a bucket cannot exceed its starting balance plus growth. The loop terminates deterministically; it cannot spin forever or burn unbounded tokens.

8 Preventing overdraw, specifically

Three independent mechanisms stop the model from spending money that isn't there:

Mechanism	Where	Effect
Budget cheat-sheet	Context (§3)	Tells the model up front each bucket's start balance, sustainable draw, and full spend-down rate — plus the current depletion map.
Hard rule in prompt	System prompt (§4)	“A bucket can supply at most its starting balance plus growth; total lifetime withdrawals must stay within that budget.”
Simulated truth	Dry-run + refine (§6–7)	The engine reports actual $0 events from real balances; depletion is a fact, not the model's estimate, and triggers revision.

Crucially, “overdraw” is defined by the simulation, not by the model's arithmetic: a bucket's per-phase ending balance going `<= 0` sets its depletion flag. Because verification uses the same `calcAllPhases` as the rest of the app, a proposal that passes here behaves identically once applied — no drift between “verified” and “live”.

9 Guardrail summary & human-in-the-loop

The defences are layered — defence-in-depth, so no single failure is catastrophic:

#	Guardrail	Stops…
1	Bounded field namespace (s.* / pN.*)	The model touching anything outside known plan inputs.
2	Numeric-only structured output	Vague or unactionable 'changes'; everything is simulable.
3	Context headroom + budget facts	Proposals that ignore tax/ACA/IRMAA limits or bucket budgets.
4	Prompt: never break a passing check	Trading a fixed goal for a newly-broken constraint.
5	Deterministic dry-run	Outcomes being taken on the model's word.
6	Bounded refine loop (max 3)	Persisting a depleting plan; runaway iteration.
7	Verification badge	The user approving an unverified change unknowingly.
8	Opt-in apply (per-row checkboxes)	Anything reaching live state without explicit consent.

The final gate is human. `showAiProposals()` renders a card where every line is a checkbox (default checked) showing `current → proposed` and the rationale, topped with the verification badge. `aiApplyProposals()` writes **only the checked rows** to `s` (and the matching DOM inputs), then the plan recomputes. Nothing is

auto-applied.

10 Determinism, privacy & failure modes

Determinism

Every figure the AI is shown or verified against is reproducible. The Monte Carlo simulation is seeded from a hash of the plan inputs (`mulberry32`), so the same plan always yields the same numbers — the model never receives drifting inputs that could flip a recommendation between identical runs.

Privacy & transport

The app is a single offline file. The user's Anthropic API key is stored only in their browser; calls go directly from the browser to the API (`anthropic-dangerous-direct-browser-access`). Outgoing messages are sanitised to `{role, content}` only, so local display-only fields (e.g. the Save-PDF question label) never leave the device. No server, no telemetry.

Failure modes (all graceful)

- **No / exhausted credits, bad key, rate-limit, timeout** — mapped to a friendly message via a shared `_aiFriendlyError()`; the entire deterministic planner keeps working without AI.
- **Malformed proposal JSON** — the parse is wrapped in try/catch; a bad block is ignored, leaving the prose reply intact.
- **Un-simulable clone** — if the dry-run can't run, it returns null and the proposal is shown without a false 'verified' claim.
- **Plan edited mid-chat** — fingerprint mismatch injects a 'plan updated' note and trims stale history.

11 Extension ideas & hardening

For anyone extending the engine, natural next steps that fit the existing grain:

- **Schema-validate proposals** before simulating — whitelist `field` against the known namespace and reject unknown keys explicitly (today an unknown `pN.field` is a harmless no-op write to the clone; an allowlist would make that intent auditable).
- **Clamp proposed values** to sane ranges (non-negative withdrawals, ssStartAge ∈ 62–70) as a pre-sim filter, so an out-of-range hallucination is corrected before it reaches the engine.
- **Generalise the refine trigger** beyond depletion — e.g. auto-refine when a proposal would break a passing health check, reusing the same `SIMULATION RESULT` feedback pattern.
- **Per-bucket budget assertions** in the dry-run result (surface remaining headroom per bucket) to give the model even tighter feedback on revisions.
- **Tool-use / structured-output API** — migrate the <proposals> convention to the provider's native structured-output or tool schema for stricter typing, while keeping the deterministic verify-and-apply core unchanged.

Takeaway. The model is a fluent intent generator wrapped in a deterministic safety cage: bounded in what it can propose, grounded in real plan facts, fact-checked by the production engine, corrected against ground truth, and gated behind explicit human approval. Suggestions are cheap and creative; consequences are computed by code.