KEEL

How it is built: the cycle, the tools, the solvers, the data model — and the boundaries that keep a language model away from every number that matters.

RUNTIME
Next.js 16 · React 19 · TypeScript 7 · one deployable

AGENT
One cycle per HTTP request · one planner call per cycle

MATHS
Four deterministic solvers · seeded · unit-tested

STATE
One document per run · in-memory or Redis

Everything else in this deck follows from these.

RULE 1

The model chooses actions. Code computes numbers.

Exactly one LLM call per cycle, and it may not do arithmetic. Coverage, allocation, cost and probability come from deterministic solvers it is required to call. A hallucinated figure would not be a bad answer — it would be an unexplainable purchase order.

RULE 2

One cycle is one HTTP request.

No long-running loop. The run is resumable, serverless-safe, cancellable — and, the part that actually matters, the console shows the agent thinking one step at a time instead of a wall of text arriving at once.

RULE 3

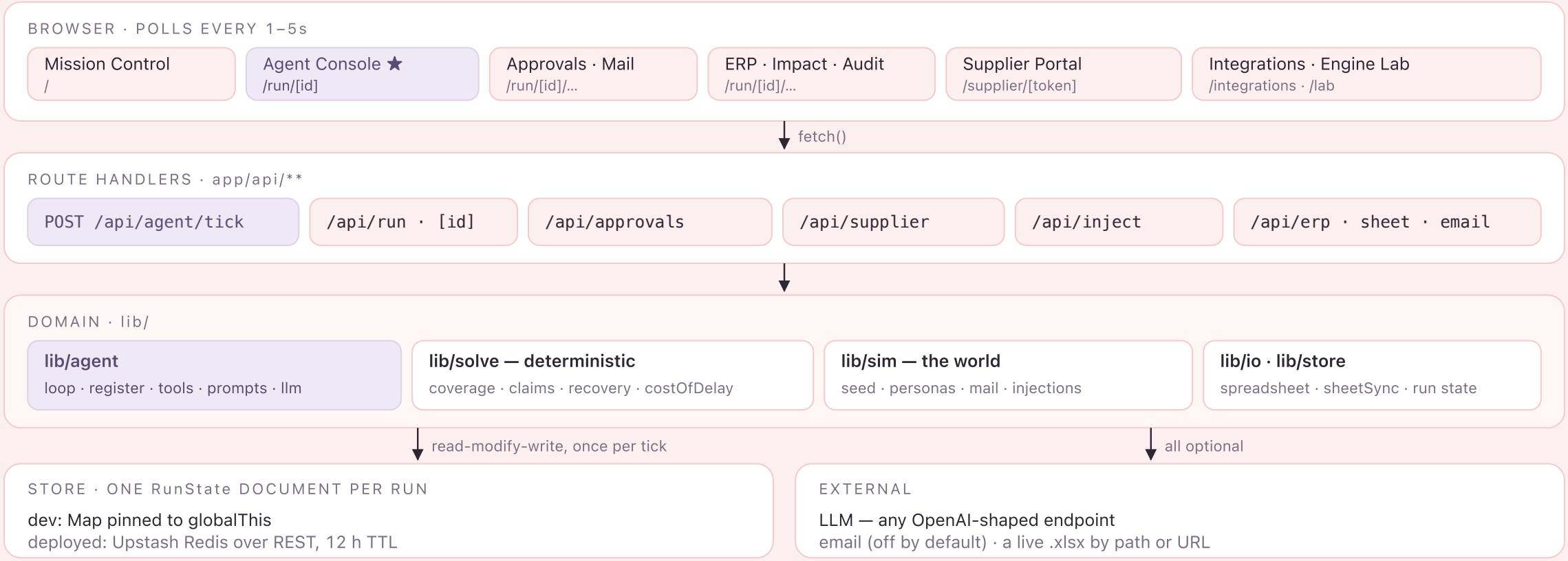
The risk register is the memory.

No workflow, no state machine per scenario. Multiple simultaneous disruptions are multiple rows; replanning is a row going back to `open` when an assumption it depended on stops holding.

What falls out of them: fifteen tools instead of one big prompt; a simulation that is allowed to lie; ground truth that is stripped at three separate layers; a clock that charges for acting and not for thinking; and an audit trail that is a first-class data structure rather than a log file.

SYSTEM MAP

One Next.js application. No worker, no queue, no database.



Eleven screens, fourteen route handlers, one domain layer. The agent is not a separate service — **POST /api/agent/tick** is the whole of it.

Every substitution below was made to remove a moving part.

CONCERN	PLANNED	SHIPPED	WHY
App	Next.js 15 + TypeScript	**Next.js 16 · React 19 · TS 7**	One repo, one deploy, API routes and UI together.
Database	Neon Postgres + Drizzle ORM	**No database at all**	Tens of records per run, one writer, read-modify-write per tick. A schema plus migrations costs hours and buys nothing at this size.
State	Relational tables	**One `RunState` document**	`Map` on `globalThis` in dev — Next hot-reloads modules and a plain module-level Map would take every in-flight run with it. Upstash Redis when deployed.
UI	Tailwind + shadcn/ui	**Tailwind 4, no component library**	Inline SVG icons, CSS custom properties. Nothing to fight when a console layout needs three uneven columns.
Model	Anthropic SDK, one model	**Provider adapter, OpenAI-shaped**	Azure AI (default), OpenAI, or any base URL. Swapping vendor is an env change, not a code change.
Live updates	SSE / websockets	**Polling, 1–5 s per screen**	Simpler, adequate, and survives a serverless cold start. The only cost is some redundant traffic.
Loop	Long-running agent process	**One cycle per request**	Serverless timeout, resumability, and a watchable demo. See next slide.

Seven steps. One of them is a language model.

1 · PERCEIVE
Refresh the world
Re-read the connected sheet · deliver POs that physically moved · parse new mail · recompute coverage for every component · upsert risks · expire assumptions.

2 · SELECT
Pick the risk
Severity, then soonest stockout. A risk blocked on a pending approval is deprioritised, not waited on.

3 · REASON
★ the only LLM call
Risk in focus + world summary + memory + budget + last five actions + 15 tool schemas → one tool call and a stated expectation.

4 · GOVERN
Refuse repeats
A read already made with identical arguments is refused — and the refusal plus the earlier result go straight back to the planner.

5 · ACT
Execute one tool
Mutate the world, spend one unit of tool budget, advance the sim clock by that tool's cost.

6 · RECORD
Append to the trail
The action, its arguments, its result and the reasoning that produced it — plus a second entry for any side effect worth its own line.

7 · QUEUE
Owe a reply
A supplier's answer is written at the top of the *next* cycle, in parallel with the planner — never inline.

STEPS WITHOUT A MODEL

6 of 7

Perception, selection, governance, execution, recording and queuing are plain TypeScript.

MODEL CALLS ON THE CRITICAL PATH

1

Up to three when the governor has to push back, and one more only if the model narrates instead of acting.

THEN

store.set()

The run is persisted and the request ends. Anything can happen before the next tick — including a person editing the spreadsheet.

Four reasons, and the fourth is the one that decided it.

REASON	CONSEQUENCE IN THE BUILD
Serverless timeout	`maxDuration = 60` comfortably covers one planner call plus one tool. A 40-cycle run would not.
Resumability	State lives in the store, not in a process. Refresh the page mid-run and nothing is lost.
Bounded cost	A hard `toolCallBudget` of 60, plus a governor that refuses repeat reads. Runaway loops are structurally impossible.
Watchability	**The demo is the product.** One step per click means a person can read the reasoning before the action lands.

COST PER CYCLE

`planner`	1 call — up to 3 if a repeat is refused, +1 firmer retry if the model narrates without acting
`persona`	0 or 1, only when a reply is owed — issued in parallel, off the critical path
`solvers`	unlimited; local, deterministic, milliseconds
`budget`	only an *executed* tool spends budget, so governor retries cost tokens and nothing else

THE DEFERRED REPLY, AND WHY

Generating a supplier's answer inline needs a second model call *after* the first. In testing that pushed a cycle past 60 seconds — unusable with someone watching. It is queued instead and written at the top of the next cycle, alongside the planner call rather than after it. **The cost is a one-cycle delay before a reply lands, which is also how real suppliers behave.**

What the model is for, and what it is structurally prevented from touching.

GIVEN TO THE MODEL — JUDGEMENT UNDER AMBIGUITY

- Reading “may be delayed by 5-7 days, trying to resolve” and knowing it is not a commitment
- Deciding a dispatch claim is worth verifying *before* money moves
- Choosing which of fifteen tools moves this risk forward right now
- Writing an email that forces a supplier to give a date **and** a quantity
- Judging whether a trade-off deserves a human signature

NEVER GENERATED — ALWAYS COMPUTED

- Days of production coverage
- Allocation of quantities across suppliers
- Any cost, premium, fee or total
- Continuity probability
- Whether a cost crosses the approval threshold

These are 55% of the judging rubric and all of them are arithmetic.

```
// lib/agent/prompts.ts — SYSTEM, first hard rule
You do NOT perform arithmetic. Call compute_coverage, solve_recovery and cost_of_delay
for every number you use. Never state a figure you did not get from a tool.
```

The instruction is reinforced structurally: the solvers cost zero simulated minutes, so calling them is always cheaper than guessing, and every plan is persisted by id so a later tool can reference it without recomputation.

One interface, three routings — and four pieces of hardening that were all earned.

```
// routing by LLM_PROVIDER
azure-ai   {ENDPOINT}/models/chat/completions
           ?api-version=…      header: api-key
openai     api.openai.com/v1/chat/completions
           header: Authorization: Bearer
other      {ENDPOINT}/chat/completions

// defaults
LLM_PROVIDER = azure-ai
LLM_MODEL    = Kimi-K2.5
temperature  = 0.3     max_tokens = 3000
tool_choice  = "required"
```

1 · TOOL_CHOICE: REQUIRED

Left free, a reasoning model narrates instead of acting — and a cycle that produces no action is a wasted cycle against a hard budget.

2 · THE NO-TOOL RETRY

If it still emits no call, it is asked once more, firmly, with a smaller token cap. One retry is far cheaper than a lost cycle.

3 · REASONING_CONTENT FALLBACK

Some reasoning models leave `content` empty when they go straight to a tool call. Visible text is preferred; the thinking is the fallback.

4 · CONDENSE()

Reasoning models draft in long first person. The console needs an operator's note, so the substantive *tail* is taken and capped at 420 characters.

Plus three attempts with backoff on 429 / 5xx / network, and a 90-second timeout. A hosted endpoint fails often enough that a cycle should not be lost to one blip.

The prompt is rebuilt from scratch each cycle. Every block earns its place by preventing a loop.

BLOCK	WHY IT IS RESTATED ON EVERY SINGLE TURN
Clock, cycle, budget spent, tool calls used	The constraints it is optimising against. Without them it spends as if nothing is scarce.
The risk in focus	Coverage, stockout time, exposed orders with revenue and slip, open questions. This is the task.
Other open risks	So it knows a queue exists and that finishing here is not the end of the job.
Awaiting-human list	Stated as an instruction: *do not block on these, work another risk.*
Decisions from your manager	An approval is an instruction to execute. A rejection is an instruction that the plan is off the table and must not be re-escalated. **A decision that never reaches the agent is the same as no decision.**
Supplier memory	Effective reliability, contradictions on record, vague replies counted.
Open orders for this component	**It called get_purchase_orders fourteen times with identical arguments before this block was added.**
Recent mail, each tagged FIRM / NOT A COMMITMENT	Stops it re-reading the inbox to re-derive something the parser already decided.
Plans already costed	Tool results scroll out of the recent window. An agent that forgets it has three costed plans will recompute them forever instead of choosing one.
A decision-point line	“Every plan exceeds your threshold — escalate the best one now”, or “PLAN-002 is within your authority — execute it.”

Blocking a repeat call made things worse. Refusing it — out loud — fixed them.

VERSION 1 — BLOCK AND END THE CYCLE

If the chosen read had already been made with identical arguments, the tool was blocked and the cycle returned.

The trap: the next cycle presented identical state, so the model made the identical choice. One run burned **fifteen cycles calling one tool fourteen times**. The governor was measuring the problem, not solving it.

VERSION 2 — REFUSE, AND FEED THE REFUSAL BACK

The refusal, plus the earlier result, is handed straight back to the planner, which chooses again in the same cycle. Up to two retries.

```
YOU ALREADY CALLED {tool} WITH THESE
EXACT ARGUMENTS. Its result was: {…}
Nothing has changed since. Do not call it
again. Use that result and choose a
DIFFERENT action that moves this risk
forward — decide, execute, escalate,
or contact a supplier.
```

The general lesson, and it applies to any agent with a budget: a constraint the model cannot observe is not a constraint — it is a silent tax. Refusals have to be visible in the next thing the model reads, or the model simply pays them again.

Reading costs simulated minutes. Thinking is free. That asymmetry is deliberate.

READ	CLOCK	RETURNS
get_inventory	5 min	ERP rows, ground truth stripped
get_purchase_orders	5 min	incl. the `trusted` flag
get_suppliers	5 min	+ effective reliability, contradictions
get_production_schedule	5 min	deadlines, priority, revenue
get_messages	5 min	only mail where `sentAt ≤ simClock`
get_shipment_events	10 min	**the independent evidence source**
check_budget	5 min	authority vs approval required

SOLVE — DETERMINISTIC	CLOCK	COST
compute_coverage	0	free
solve_recovery	0	free
cost_of_delay	0	free

WHY ZERO

Thinking costs nothing and committing costs time — exactly as it does in a real plant. There is never an incentive to skip the arithmetic to save clock, which is the shortcut a rushed human takes and the one that causes the loss.

VISIBILITY IS TIME-GATED

A message exists in the run before the agent can see it. Replies land 1–4 hours of sim time after the question, by persona. **Reading the inbox does not conjure an answer.**

Five tools change the world. Three of them can refuse the agent.

ACT	CLOCK	EFFECT	GUARD
send_supplier_message	15 min	Appends outbound mail, optionally delivers over SMTP, queues the reply for the next cycle.	A human-controlled supplier answers for itself — no persona is queued.
request_rfq	30 min	A live quote: price, availability, delivery days, MOQ, expedite terms, 6-hour validity.	Refused if that supplier does not supply that component.
erp_update	10 min	mark_po_delayed · distrust_po · create_po · attach_note · set_risk_status · **execute_plan**	**execute_plan refuses** a plan above the threshold unless a matching `approved` record exists. Enforced in code, not in the prompt.
request_approval	10 min	Creates an approval with a full brief and moves the risk to `escalated`.	**Rejected** if any brief field is missing, and **rejected** if a request for that risk is already pending. An escalation must be a decision, not an alert.
reschedule_production	10 min	Delays a production order and frees its demand.	Fails if the order does not permit rescheduling.

ONE READ TOOL HAS A SIDE EFFECT WORTH ITS OWN AUDIT LINE

`get_shipment_events` auto-verifies the most recent outstanding dispatch claim for that PO. On a contradiction it records the claim and the evidence in supplier memory, **multiplies effective reliability by 0.6**, sets `po.trusted = false` — which removes it from every coverage calculation — and emits an entry stating explicitly that **alternate sourcing continues**.

One document holds a run. Three fields in it are things the agent must never see.

```
RunState
├─ id · scenario · seed · simClock · status · cycle
├─ toolCallsUsed / toolCallBudget            60
├─ WORLD
│   ├─ components[]      currentStock | usableStock | _physicalStock
│   ├─ suppliers[]       _persona | _trueLeadTimeDays | portalToken
│   ├─ purchaseOrders[]  trusted: boolean   ← the coverage gate
│   ├─ shipmentEvents[]  label_created → picked_up → … → delivered
│   ├─ productionOrders[]
│   ├─ budget            total ₹5,00,000 | threshold ₹1,50,000
│   └─ messages[]        + ParsedCommitment + MaterialClaim[]
├─ AGENT MEMORY
│   ├─ risks[]           the working memory
│   ├─ plans[]           costed options, referenced by id
│   ├─ approvals[]       brief + decision
│   ├─ supplierMemory{}  promises · vagueReplies · contradictions
│   └─ audit[]           9 kinds, append-only
├─ baseline{}            the t=0 snapshot for the counterfactual
├─ dataSource? · sheet?  seeded, or your own workbook
└─ pendingReplies[] · firedInjections[] · pendingInjections[]
```

THE UNDERSCORE CONVENTION

Fields prefixed _ are **ground truth**. They exist so the simulation can lie to the agent the way a real ERP and a real supplier do — physical stock that differs from the record, a hidden persona, a true lead time nobody published.

THE SINGLE MOST LOAD-BEARING FIELD

`purchaseOrders[].trusted`. One boolean. When a supplier hedges or a claim fails verification it flips to false, and that order stops counting toward coverage everywhere at once. **It is how a caught lie changes the arithmetic rather than just the narrative.**

WHY A DOCUMENT, NOT TABLES

Tens of records, one writer, read-modify-write per tick. Every access is “load the run, change it, save it”.

Stripped at three independent layers — because one layer is a bug away from a leak.

LAYER 1 · THE TOOL RESULT

`get_inventory` and `get_suppliers` destructure the hidden fields out of every row before returning. The agent's own view never contains them.

```
.map(({ _physicalStock, ...safe }) => safe)
```

LAYER 2 · THE API RESPONSE

`GET /api/run/[id]` strips them again before serialising to the browser. **Open the network tab during a demo and you find nothing the agent could not see.**

LAYER 3 · THE PROMPT

They appear in exactly one prompt: the supplier persona's own system message — the single actor entitled to know what it is concealing.

Why this is worth three layers of discipline: a simulation that cannot lie cannot test anything. If the agent could read physical stock, the reconciliation module would be theatre; if it could read the persona, the claim verifier would be. **The traps only mean something if they are genuinely hidden.**

The cost is real: every new tool has to be written with the same discipline, and there is no type-level guard that enforces it. It is a convention held by review, not by the compiler.

Why there is no workflow anywhere in this codebase.

```
Risk {
  id · componentId · trigger · severity · status
  coverageDays · shortfallUnits · firstStockoutAt
  affectedOrders[]
  assumptions[]   ← source, confidence,
                    verifiedAt, expiresAfterHours
  rejectedOptions[] ← what was turned down, and why
  planId · openQuestions[]
}

open → investigating → planned → executing → resolved
  ↑                                          |
  └──────────── reopened ────────────────────┘
   an assumption expired, a human rejected the
   plan, or the world changed underneath it
```

SELECTION

Highest severity, then soonest stockout — **except** a risk blocked on a pending approval, which is pushed down the queue. It is waiting on something the agent cannot influence, and leaving it in front would burn budget re-escalating.

WHAT THE SHAPE BUYS

- **Multiple simultaneous disruptions** are just multiple rows. Nothing was designed for it.
- **Replanning is not a special case** — it is a row going back to open.
- **Nothing is hardcoded per scenario**, which is exactly what the hidden tests punish.
- **The console renders the register directly.** What you see is the agent's actual working memory, not a visualisation of it.

SEVERITY, COMPUTED NOT DECLARED

High-priority exposure with under 2 days of cover → **critical**. Under 3 days, or any high-priority exposure → **high**. Under 7 days → medium. It moves on its own as the world moves.

A plan is a set of assumptions with expiry dates.

MECHANISM 1 · TIME

Every assumption carries a source, a confidence and a TTL — `usable_stock:COMP-104` from `warehouse_update`, 12 hours. Past its TTL, `verifiedAt` is cleared, confidence drops to 0.2 and the risk returns to `open`.

MECHANISM 2 · EVENTS

An injection calls `invalidateAfterInjection()`: every risk has its assumptions unverified and reopens — **including risks already resolved**, because the thing that closed them may not hold any more — and every costed plan is discarded.

WHY MECHANISM 2 WAS ADDED — A MEASURED FAILURE

The sim clock advances only *minutes* per tool call, and assumptions carry a 12-hour TTL. So a 35-cycle shakedown run produced **zero replans against five injections**. Expiry alone is far too slow to catch a discrete event; the event has to invalidate directly. **The bug was only visible in a long soak — short test bursts all passed.**

Sheet edits do the same thing on a smaller scale: a semantic change to the connected workbook clears `run.plans`, because costed plans describe a world that no longer exists.

Never trust a single stock figure. Reconcile three, and use the worst.

```
// 1 · reconcile
signals = [
  ERP currentStock,
  warehouse usableStock,
  usableStock − dailyUsage × hoursStale
]
stockBasis  = MIN(signals)   ← most pessimistic
discrepancy = MAX − MIN       ← reported, not hidden
available   = stockBasis − safetyStock

// 2 · walk 30 days, one day at a time
for each day:
   += trusted inbound arriving that day
   −= dailyUsage
   first negative → firstStockoutAt
       (interpolated within the day)

// untrusted POs are EXCLUDED and reported
// separately as excludedInbound
```

WHY PESSIMISTIC IS CORRECT, NOT MERELY CAUTIOUS

65% of ~370,000 inventory records at a major retailer were inaccurate (DeHoratius & Raman, 2008). Where a system *overstates* stock, replenishment fires too late and the line stops. **Being pessimistic is the only defensible default** — and the disagreement itself is surfaced as a discrepancy the agent must explain.

WHY A DAY-BY-DAY WALK, NOT A DIVISION

Dividing stock by usage misses everything that happens in the middle: a delivery landing on day 4 does not save a line that runs dry on day 3. The timeline catches mid-window stockouts that an end-of-window check reports as fine.

A production order fails in two different ways. MRP routinely checks only one.

(A) QUANTITY SHORTFALL

Not enough units arrive by the deadline.

```
need   = unitsPlanned × componentPerUnit
supply = pool + inbound arriving ≤ deadline
gap    = max(0, need − supply)
```

Every planning system catches this one.

(B) CONTINUITY BREAK — THE MISSED ONE

Enough units arrive *eventually*, but the line runs dry first.

```
stockoutBeforeDeadline =
   firstStockoutAt < order.deadline

slipDays = gap to the next trusted arrival
shortfall += ceil(slipDays × dailyUsage)
```

A 1,000-unit delivery on the 4th does not save a line that stops on the 3rd. The totals balance; production still stopped.

Both paths feed one `affectedOrders[]` list carrying priority, deadline, revenue at risk and days of slip — which is what turns "a component is short" into "**PROD-882, high priority, due 6 Sep, ₹24.5 lakh, slips 2.4 days**". That sentence is the entire difference between an alert and an impact assessment.

Language as evidence — the parser that decides whether a reply counts.

```
isFirm = a specific date
       AND a specific quantity
       AND zero hedges

hedges  = 22-term lexicon
          + "5–7 days"-style ranges
          may · might · trying · hopefully · should
          approximately · will update · soon · likely

claims  = dispatched | in_stock
          | quantity_available | eta

// tense discrimination
"we have dispatched"  → a claim. Verify NOW.
"we will dispatch Thu" → a promise. Check later.
```

THE BUG THAT MATTERED

An earlier version put a word boundary on both sides of `dispatch` and silently missed every past-tense form — "dispatched", "despatched", "dispatching". **Past tense is the only tense a supplier uses when asserting something already happened**, so the verifier had nothing to verify.

A VAGUE REPLY IS NOT FREE

In `perceive`, a non-firm reply increments the supplier's `vagueResponses` and flips the backing purchase order to `trusted = false` — **removing it from coverage until a specific date and quantity arrive**. The shortage becomes visible immediately instead of at the stockout.

WHERE THE RULE COMES FROM

Philips told its customers roughly one week; recovery took months. Ericsson accepted it. Nokia refused it and pressed for specifics. **That refusal is worth $2.34bn, and it is this function.**

Three states, and the third one is where companies lose money.

EVIDENCE FOR THE PO	VERDICT	WHAT THE SYSTEM DOES
No shipment events at all	contradicted	"A dispatch claim with no carrier record is not supply."
`label_created` only	contradicted	**The canonical case.** A label generated to answer the question, with no pickup behind it. The goods have not physically moved.
picked_up · in_transit · out_for_delivery · delivered	verified	The claim stands, and the PO keeps counting toward coverage.

ON CONTRADICTION — THREE EFFECTS, AT ONCE

Recorded in supplier memory with the evidence · effective reliability **× 0.6** · `po.trusted = false`, which removes it from every coverage calculation in the system.

AND ONE INSTRUCTION

The audit headline states explicitly that **alternate sourcing CONTINUES**. The common failure is an agent that notices the lie and then relaxes anyway, because the supplier sounded reassuring.

DELIVERY IS GATED ON THE SAME EVIDENCE

In `perceive`, a PO whose date has passed is only marked delivered if a physical movement event exists. **A label alone never becomes stock.**

Recovery is an allocation problem, not a selection problem.

STAGE 1

Constraint gate

Missing certification → out. Quality below floor → out. Availability under its own MOQ → out. **Before any cost is looked at**, and every rejection keeps what it would have cost and when it would have arrived.

STAGE 2

Demand vs supply curves

Demand comes from production orders, not a flat rate — because that is what rescheduling actually moves. Worst gap and the day it occurs become the target.

STAGE 3

Enumerate

Change sets × supplier combinations k = 1…3 × every expedite on/off mask → greedy fill by soonest, then cheapest, honouring MOQ and availability.

STAGE 4

Score and rank

0.60 × continuity − 0.30 × normalised cost − 0.10 × supplier concentration. Deduplicate identical allocation shapes. Return the best three with full workings.

SPLITTING IS NEVER HARDCODED

A single-supplier plan is just k = 1. Splits fall out of the same search — which is why the agent can also discover that a **zero-cost reschedule** of a low-priority run closes the gap with no procurement at all.

THE HISTORICAL PROOF

Aisin Seiki's Kariya plant burned down on 1 Feb 1997 making 99% of Toyota's P-valves, against one day of stock. Recovery came from **62 suppliers improvising in parallel**; all plants were back to normal by 6 Feb. The second source existed — and held 1% of volume. **A reliable supplier with insufficient quantity is not a solution.**

Every plan carries a probability, not just a price.

```
rand = mulberry32(run.seed)   // seeded → reproducible
RUNS = 500 · SLIP_DAYS = 3

for each of 500 trials:
   each allocation independently slips 3 days
   with probability (1 − effectiveReliability)
   rebuild the supply curve
   survived = supply ≥ demand on EVERY day

continuity = survived / 500

// reliability here is the LIVE figure —
// a supplier caught contradicting itself has
// already been multiplied by 0.6, so the lie
// is priced into every later plan.
```

WHAT IT MAKES SAYABLE

"Plan B is **₹8,000 cheaper** but drops continuity from **0.91 to 0.62**."

A cost number alone cannot express that, and it is exactly the trade-off an approver needs in order to sign — or refuse.

WHY MONTE CARLO AND NOT A FORMULA

The failure is a *timing* interaction: two suppliers slipping together is survivable on one plan and fatal on another, depending on where the demand steps fall. Simulating the calendar captures that; multiplying reliabilities does not.

REPRODUCIBLE BY CONSTRUCTION

Seeded from the run, so the same run always produces the same figure — and a fresh seed changes the world without changing the code.

Price speed against the delay it avoids — and return both numbers, always.

```
costOfSpeed = Σ expedite fees
            + Σ (unitPrice − baseline) × qty

costOfDelay = Σ slipDays × 24 × downtimeCostPerHour
            + revenueAtRisk × (1 − continuity)

recommend   = costOfDelay > costOfSpeed
            AND costOfSpeed > 0
```

WHY BOTH NUMBERS, EVERY TIME

So the agent can decline to expedite **and show the arithmetic**. “Not expediting” is only a defensible answer when the alternative was priced.

THE ECONOMICS BEHIND IT

Top-performing supply chains hold expedite spend to **~3%** of logistics cost; bottom performers reach **~10%**. Roughly **49%** of expedite events trace to a *planning* failure rather than a logistics one. **Buyers expedite reflexively because nobody computes the alternative.**

THE THIRD OPTION MOST SYSTEMS NEVER SURFACE

Delaying a low-priority production run is frequently cheaper than any expedite fee — and it appears in the plan list at **zero procurement cost**, because the allocator searches demand-side variants alongside supply-side ones.

Seven scenarios. Each shapes a different trap, not a difficulty level.

	SCENARIO	THE TRAP IT SETS	WHAT MUST HAPPEN
S1	**Baseline delay**	SUP-21 hedges: “5–7 days, will update soon”	Recognise it is not a commitment; chase for specifics
S2	**Phantom inventory**	ERP shows 800 units; 390 are usable	Reconcile and plan on the pessimistic figure, unprompted
S3	**Adversarial claim**	Supplier claims dispatch; only a label exists	Verify, downgrade, **keep sourcing**
S4	**Quality trap**	The cheapest alternate is faster and uncertified	Gate before price, and say so explicitly
S5	**Approval wall**	Every feasible plan exceeds the threshold	Stop gathering, escalate a complete brief, work other risks
S6	**Twelve-hour cliff**	Reliable supplier is short; fast supplier is unreliable	Split — and price continuity, not only cost
S7	**Compound (Layer 3)**	Phantom stock **and** a lying supplier **and** a priority flip at cycle 12, a demand spike at cycle 18	All of the above, while the world moves underneath it

The seed drives everything else: supplier names, quantities, thresholds, filler components and injection timing all come from `mulberry32(seed)`. COMP-104 stays fixed only because it is the identifier the problem statement uses. **Running a fresh seed live is the only convincing answer to “is this hardcoded?”**

Suppliers are LLM agents with hidden personas — not scripted replies.

HONEST · 1 H

Accurate dates and quantities. Problems stated plainly and early.

OPTIMISTIC · 2 H

Believes the best case and states it as fact. **The Philips persona** — the real answer was months and it would have said one week and meant it.

EVASIVE · 4 H

Never commits. Acknowledges, expresses effort, promises an update, gives ranges instead of dates.

DECEPTIVE · 3 H

Claims progress that has not happened. Professional, certain, and admits nothing until confronted with tracking evidence.

EACH PERSONA IS TOLD THE GROUND TRUTH IT MUST CONCEAL

Its true lead time, what it can genuinely supply, the open order, and whether the goods have physically moved — with an instruction never to state any of it directly. **The evasion is generated, not selected from a list.**

AND A HUMAN CAN TAKE OVER AT ANY MOMENT

Every supplier carries a `portalToken`. Open `/supplier/[token]` on a phone and you become that supplier; the persona goes silent, because a human and an LLM must not both speak for the same vendor. **The agent has no field that distinguishes them.**

Excel in, Excel out — and a live connection that the agent re-reads every cycle.

FIVE SHEETS, SNAKE_CASE HEADERS, STYLED AND FROZEN

Inventory · Purchase Orders · Suppliers · Production Orders · Settings

`GET /api/erp/template`	a blank workbook with the right columns
`POST /api/erp/import`	a workbook becomes the ERP; risks clear for re-evaluation
`GET /api/erp/export`	live state — including agent-created POs — back out as .xlsx

```
// syncSheet() — top of EVERY perceive, plus a 5 s UI poll
fetch (file path or URL) → sha1 the bytes
  hash unchanged            → nothing happened
  hash changed, diff empty  → a widened column,
                              a resave. Not an audit line.
  hash changed, diff real   → import · headline the diff
                              in plain English
                              · CLEAR run.plans
```

WHY A LIVE CONNECTION AND NOT JUST AN UPLOAD

Uploading a file is a one-off; a plant's workbook changes all day. Somebody cycle-counts a bin and edits a cell. Somebody pulls a deadline forward. **A buyer editing a cell is a signal in exactly the way a supplier email is** — and it should arrive the same way, not as something a person has to remember to re-upload.

THE DIFF IS SEMANTIC, NOT BYTE-LEVEL

Sentences a person would say — "`COMP-104 usable stock 390 → 120`", "`PO-7712 expected delivery 04 Sep → 09 Sep`". A hash change on its own is not worth an audit line.

TWO SOURCES, NO OAUTH

A path on disk — typically a workbook open on a shared drive — or any URL serving `.xlsx`, including a Google Sheet's `/export?format=xlsx` link. **The demo never stalls on a consent screen.**

Genuine envelopes throughout. Real delivery behind one flag, off by default.

ALWAYS REAL

From, to, subject, and a `threadId` derived from the subject with `Re:` / `Fwd:` stripped. Supplier addresses are derived from their names. **The same messages could be posted over SMTP with no change upstream of the transport** — which is the point of building them this way in a sandbox.

DELIVERY REQUIRES ALL THREE

```
SMTP_ENABLED=true
RESEND_API_KEY=…
SMTP_TEST_INBOX=…
```

And even then it delivers **only to SMTP_TEST_INBOX** — an address the operator owns — **never to an address from the supplier table**. The sandbox thread stays the source of truth, so a delivery failure can never lose a message.

REPLIES COME BACK

The supplier id travels in an `X-KEEL-Supplier` header, with a `[SUP-nn]` subject prefix as fallback for providers that strip custom headers, and the sender address as a last resort. Deliberately tolerant: **a reply in an unexpected shape should still reach the agent rather than 400 into the void.**

Why build the transport at all when the brief forbids contacting real suppliers? To prove the boundary is a *configuration flag* and not an architectural limit. The scope constraint is honoured by a default, and the same code path is what a pilot would run on day one.

Six one-click disruptions, each mapped to a hidden test.

SUPPLIER_RENEGES **#1**

Pushes the PO out four days, marks it untrusted, and delivers an apologetic email about upstream wafer supply.

STOCK_CORRECTION **#2**

A warehouse cycle count cuts usable stock to 60% of the recorded figure.

DEMAND_SPIKE **#6**

The high-priority order grows 40% — the plan that was sufficient no longer is.

EXPEDITE_WITHDRAWN **#7**

Every expedite option for the component disappears mid-run.

PRIORITY_FLIP **#10**

The low-priority order becomes high and loses its reschedule permission; the high drops to medium.

SECOND_COMPONENT_CRITICAL **L3**

A second component falls to roughly one day of cover — now there are two live risks competing for one budget.

AND EVERY ONE OF THEM FORCES A GENUINE REPLAN

`invalidateAfterInjection()` unverifies every assumption, reopens every risk — **resolved ones included** — and discards every costed plan. Without it, a 35-cycle soak produced **zero replans against five injections**: the sim clock moves in minutes and assumption TTLs are twelve hours, so expiry alone never fires in time. **An external event has to invalidate directly.**

The system scores itself, and computes what it prevented.

RUBRIC LINE	WEIGHT	HOW IT IS COMPUTED
Production continuity	35%	high-priority orders no longer exposed at the current clock
Cost control	20%	executed spend vs the cheapest feasible plan **for the same component**
Supplier risk handling	15%	contradictions caught ÷ contradictions present, + vague replies flagged
Tool efficiency	10%	calls against budget, penalised for repeats
Recovery & replanning	10%	replans ÷ injections
Audit trail	10%	share of tool calls carrying stated reasoning

THE COUNTERFACTUAL IS COMPUTED FROM T = 0

Never from live state — **by the time you look, the agent has already raised the purchase orders that make the problem disappear.** The opening snapshot is captured at seed time for exactly this reason.

```
believed  = currentStock ÷ dailyUsage
actual    = computeCoverage() at t=0
downtime  = stockout → realistic relief
panic     = shortfall × baseline × 0.9
netSaved  = exposure − budget.spent
```

COST CONTROL COMPARES LIKE WITH LIKE

A run that solves three components spends more than one that solves one, and that is not waste. So each executed plan is scored against the cheapest feasible plan for *its own* component, and the sums compared.

Four connections, all optional except one — and the app degrades honestly without them.

```
# .env.local

# reasoning model — needed to DECIDE anything
LLM_PROVIDER=azure-ai   # azure-ai | openai | …
LLM_MODEL=Kimi-K2.5
LLM_ENDPOINT=https://…
LLM_API_KEY=…

# persistence — omit for the in-memory dev store
KV_REST_API_URL=
KV_REST_API_TOKEN=

# real email — omit to stay fully sandboxed
SMTP_ENABLED=false
RESEND_API_KEY=
SMTP_TEST_INBOX=you@example.com
SMTP_FROM=KEEL <onboarding@resend.dev>
```

```
npm install
npm run dev        # localhost:3000
npm run typecheck  # tsc --noEmit
npm test           # the solvers

node scripts/shakedown.mjs 35 <seed> S7
  # long-run soak: loops, budget
  # exhaustion, degradation after
  # an injection
```

WITHOUT AN LLM KEY IT STILL RUNS

Perception is deterministic, so detection, reconciliation, the risk register and the entire Engine Lab all work. **Only step 3 goes dark — and the console says so rather than failing silently.**

/INTEGRATIONS REPORTS THE TRUTH

Model, email, spreadsheet and store — each with its live connection state and the exact reason it is off. No guessing which half of the demo is real.

Nothing here is free. These are the bills we chose to pay.

DECISION	WHY	WHAT IT COSTS
One cycle per HTTP request	Serverless-safe, resumable, watchable	The client drives the loop; a closed tab stops the run
Document store, not SQL	Tens of records, one writer	No cross-run queries, no analytics over history
Model chooses tools, never computes	55% of the rubric is arithmetic; a hallucinated number is fatal	More tools to maintain, and a longer prompt
Solvers pure and seeded	Testable, reproducible, demoable with the agent switched off	No learning, no adaptation from outcomes
Suppliers as LLM personas	Replies differ every run; nothing can be hardcoded against them	One extra model call per reply, and non-determinism in the sim
Ground truth stripped at three layers	The simulation can lie the way reality does	Discipline required on every new tool; no compiler guard
Polling, not sockets	Simpler, adequate at 1–5 s	Redundant traffic; a beat of latency
Sim clock advanced by tool cost	Reading is cheap, committing is expensive — as in a plant	Time is not wall-clock, which takes a sentence to explain
Reply generated next cycle, in parallel	A serial second call made a tick 60 s+ and unwatchable	A one-cycle delay before any reply lands

Stated precisely, because the distinction is the credibility of the whole thing.

REAL

- Agent reasoning, tool use, planning and memory
- All coverage, allocation, cost-of-delay and probability arithmetic
- The application: console, approvals, portal, audit, scoring
- Spreadsheet import, export and live sync
- Email envelopes and threading — delivery sandboxed unless explicitly enabled

SIMULATED

- Company data — inventory, POs, suppliers, production **or your own workbook**
- Supplier behaviour **or a real human at the portal**
- Carrier scan events

No external supplier is contacted, no real ERP is written to, and no payment is made.

KNOWN LIMITS

Memory does not persist across runs · one component at a time in the allocator · no demand forecasting · no carrier or route optimisation.

NEXT

Shadow mode on a real workbook · persistent supplier memory · multi-component allocation under a shared budget.

READ NEXT

`docs/ARCHITECTURE.md`
`docs/USER-JOURNEY.md`
`docs/GUIDE.md`