PRACTITIONER'S GUIDE

Causal Thinking with TP Studio

A practitioner's guide to the Theory of Constraints, illustrated end-to-end with TP Studio.

GENERATED 2026-07-03

TP-STUDIO.STRUKTURERETSUNDFORNUFT.DK

Contents

Foreword

> *Last reviewed against TP Studio schema v10.*

The Theory of Constraints has two literatures and one tool tradition.

The first literature is the novels — *The Goal*, *It's Not Luck*, *Critical Chain*. Goldratt taught the method the way method actually transmits: through characters, through frustration, through the slow turn from "things are just hard" to "things are this way for a *reason*, and the reason has a shape." If you have never read those books, put this one down and read them first. They will teach you why the trees in Part 2 of this book are shaped the way they are. Nothing here substitutes for that grounding.

The second literature is the academic and consulting work — *The Theory of Constraints Handbook*, the AGI white papers, Cox & Schleier's compendium. Rigorous, comprehensive, dense. The trees are explained one notation at a time. By page two hundred you have a vocabulary that works.

Between those two sit the practitioners. They have read Goldratt. They have skimmed the handbook. They have a problem at work. They want to build a Current Reality Tree on Wednesday afternoon, not next quarter. They want to know how to draw the cloud, how to verbalize the entry condition, what to do when the diagram looks wrong, when to stop. The novels teach the why; the handbook teaches the rigor; the practitioner-facing material teaches *how to actually sit down and do it*.

That third corner is what this book is for.

The tool tradition is what makes the third corner viable. Drawing a CRT by hand teaches you a lot, but it doesn't scale past the first few revisions. Drawing one in PowerPoint is a special kind of hell. The TOC software ecosystem developed over the last two decades has given practitioners a canvas; a body of practitioner-facing writing has shown how to use it. TP Studio is an open-source, local-first canvas in that tradition; this book is a practitioner companion written specifically for it.

You will get the most out of this book if you read it with TP Studio open in another window and a real problem on your desk. The exercises do not work in the abstract. Pick a domain you understand — a system that frustrates you at work, a recurring failure mode in a project you are running, a strategic question you cannot get straight in your head — and build the diagrams alongside the chapters. By the end you will have constructed at least one Current Reality Tree and at least one Evaporating Cloud for that problem; if you have done the exercises honestly, you will know something about the problem that you did not know when you started.

The TOC philosophy holds that the people closest to a system are usually capable of analyzing it correctly, given a framework to think in. This book does not aim to teach you anything you do not already know about your own work. It aims to teach you a notation in which what you know becomes legible — to you first, then to others.

Thank you for picking this up. Now open the application and let us begin.

→ Continue to Chapter 1 — The system has a goal

Colophon

Chapter 1 — The system has a goal

> *If you skip this chapter, the rest of the book will still work mechanically. But the diagrams in Part 2 will feel like notation exercises rather than thinking. Read this chapter even if you're TOC-fluent — the vocabulary established here is what the rest of the book leans on.*

The premise

Every Theory of Constraints analysis rests on one assumption, and it is the assumption that distinguishes TOC from the rest of process-improvement.

> ***The premise:*** *Every system has a goal, and every system's performance is limited by one thing — the constraint. Improving anything else does not improve the system. Improving the constraint does.*

That second sentence is the part most people skip past on first read, and the part that matters most. TOC is a *constraint-centric* view of systems. Everything else — the diagrams, the validators, the cloud, the focusing steps — is machinery for finding and dismantling constraints.

This sounds obvious until you actually try to make the obvious version stick in a room of people who are used to thinking about averages, about local efficiency, about doing-more-of-the-good-stuff. The obvious version, when taken seriously, has consequences:

- Improving anything that isn't the constraint produces no system-level gain.
- The constraint is almost never where people think it is.
- The constraint can be physical, but more often it is policy, behavior, or measurement.
- Once you fix one constraint, a new one appears — somewhere else in the system. The work never ends; the leverage just relocates.

The Thinking Processes are the toolset for finding and addressing constraints that aren't physical — and that's almost all of them, in the modern white-collar economy. A constraint of "we don't have enough drilling machines" is something a budget solves. A constraint of "our incentive structure rewards individual heroics and punishes the documentation that would prevent them" needs a Thinking Process. That's most of what we do in real organizations, and that's the gap the trees in Part 2 are built for.

Naming the gap

A constraint is only worth analyzing because it holds some measure short of where it should be. That distance — between where a measure sits today and where you want it — is the *gap*, and every tree in this book is ultimately in service of closing one. It pays to name the gap out loud before you draw anything, so the whole analysis has an anchor to be measured against.

TP Studio gives the gap a home on the document itself. The **"Performance frame"** section of the Document panel holds two optional anchors: a **Low** note — the measure's current, unacceptable level — and a **High** note — its target or desired level. "On-time delivery at 60%" → "Reach 98% within two quarters." It's general to every diagram type, not tied to any particular tree, and it travels with the document, so whoever opens the file later sees the gap the analysis was built to close stated in plain numbers rather than implied by the diagram. Filling it in is a small facilitation discipline that keeps a long investigation honest about what "done" would mean.

Closing the loop — knowing it worked

Naming the gap is the *before*; the analysis isn't finished until you've checked the *after*. A Thinking-Process investigation makes a falsifiable promise: *eliminate this constraint and this measure will move.* The discipline is to go back, once the injections have landed, and see whether it did.

That return trip is concrete, not ceremonial:

- **Re-read the CRT.** The UDEs you started from were observable effects. Weeks later, are they still observable? Re-open the original CRT — it's a revision, or a separate tab — and walk the UDEs. The ones that have gone quiet are your evidence; the ones that haven't point at a cause you missed or an injection that didn't take.
- **Re-measure the gap.** The Performance frame's Low note was the unacceptable level you started from; the High note was the target. Put the new number beside them. "On-time delivery 60% → 98%" is either true now or it isn't, and the frame makes the comparison impossible to fudge.
- **Compare the diagrams.** Capture a fresh snapshot of reality and use side-by-side compare (Chapter 14) against the original — what the FRT predicted versus what actually happened.

A constraint that's been elevated has *moved*, not vanished — Step 5 of the Five Focusing Steps sends you back to Step 1. Closing the loop is how you find out where it went.

The Five Focusing Steps

Goldratt's canonical sequence for working with constraints. Worth keeping in mind throughout the rest of the book, because *every* thinking-process tree is in service of one of these steps.

1. **Identify** the system's constraint.
2. **Exploit** the constraint — make it work to its full current capacity before adding more.
3. **Subordinate** everything else to the constraint — local optimums that fight the constraint are net-negative for the system.
4. **Elevate** the constraint — invest, change, expand. Only when 2 and 3 are exhausted.
5. **Go back to step 1** — the constraint will have moved.

The Thinking Processes map cleanly onto these steps:

Thinking Process	Maps to	What it gives you
Current Reality Tree	Identify	The actual constraint, traced from symptoms backward.
Evaporating Cloud	Identify (the *policy* constraint)	The conflict that holds the current reality in place.
Future Reality Tree	Elevate	The system as it would be once the constraint is broken.
Prerequisite Tree	Subordinate / Elevate	The obstacles to getting from current to future.
Transition Tree	Subordinate / Elevate	The sequenced actions that dismantle the obstacles.
Goal Tree	Step 1 *before* the iteration starts	The objective the constraint is interfering with.
Strategy & Tactics Tree	Elevate (when the elevation is itself a large program)	Deployment-grade decomposition of strategy into tactics.

You will not use all of these every time. Most analyses center on a CRT, dissolve one or two clouds, draft an FRT, and stop there. The PRT and TT come out only when implementation needs sequencing. The Goal Tree is usually the *frame* you draw around the whole investigation, not a separate analysis. S&T is for larger programs — multi-team rollouts, strategy decomposition, the kind of thing you don't sit and draw on a Tuesday afternoon.

Which tree, when? — a starting map

The table above maps the trees onto Goldratt's focusing steps. But you rarely start from "which step am I on" — you start from a felt problem. This map runs the other way, from the problem in front of you to the tree that fits it:

You're staring at…	Start with	Because
Symptoms everywhere, no agreement on the cause	**Current Reality Tree** (Ch 4)	It traces the symptoms back to the one or two root causes that produce most of them.
A chronic tug-of-war that never resolves	**Evaporating Cloud** (Ch 5)	It surfaces the real conflict holding the situation in place, and the assumption you can break. In a hurry, the **Rapid 3-cloud diagnosis** gets you to the core conflict from three symptoms.
A fix you like, but you're worried about side-effects	**Future Reality Tree** (Ch 6)	It checks the injection actually delivers — and hunts the negative branches before they bite. When one risk deserves its own canvas, capture it as a standalone **Negative Branch Reservation** document.
A goal you're blocked from reaching	**Prerequisite Tree** (Ch 7) → **Transition Tree** (Ch 8)	The PRT names the obstacles and the intermediate objectives that clear them; the TT sequences the actions.
Disagreement about what "good" even means	**Goal Tree** (Ch 9)	It pins the goal and the critical success factors that have to hold for it.
A multi-team strategy that has to land operationally	**Strategy & Tactics Tree** (Ch 10)	It decomposes strategy into tactics, level by level, each carrying its assumptions.
Something that isn't a TOC shape at all	**Freeform** (Ch 11)	Argument maps, decision records, dependency sketches — the canvas without the method's scaffolding.

Most real analyses are a *chain*, not a single tree: a CRT finds the core problem, an EC dissolves the conflict under it, an FRT checks the fix. That chain — the **U-Shape** — is covered in Chapter 16 and walked end-to-end in Appendix A. When in doubt, start with a CRT.

Vocabulary you'll need

This book uses the standard TOC vocabulary throughout. Most of these words mean what you'd guess they mean; a few have specific TP Studio analogues worth pinning down once.

Term	Meaning	TP Studio analogue
UDE — Undesirable Effect	A symptom the system produces that nobody wants. The starting point of a CRT.	Entity type `ude`, red stripe.
Root Cause	A terminal cause at the bottom of a CRT — the leverage point.	Entity type `rootCause`, amber stripe.
Effect	An intermediate node — caused by something, causing something else.	Entity type `effect`, neutral grey stripe.
Injection	A change you propose to make. The hypothesis you'd test.	Entity type `injection`, emerald stripe.
Desired Effect	What the system would produce instead, after the injection lands.	Entity type `desiredEffect`, indigo stripe.
Assumption	A claim that an arrow in the diagram depends on — and that someone could plausibly challenge.	Not an entity type — an edge annotation. Added from the Assumption Well in the Edge Inspector (or press `A` on a selected edge); renders as a violet card linked to its arrow.
CLR — Categories of Legitimate Reservation	The eight discipline-checks that distinguish a good causal claim from a sloppy one.	The validator system; warnings surfaced in the inspector.
Core Driver	The root cause that ladders up to the most UDEs. The constraint, in CRT form.	The reach-badge value; the `Find core driver(s)` palette command.
EC — Evaporating Cloud	A 5-box conflict diagram showing why a chronic problem is *held in place* by a real tension between two real needs.	Diagram type `ec`.
CSF — Critical Success Factor	A must-be condition for a Goal. The middle layer of a Goal Tree.	Entity type `criticalSuccessFactor`.
NC — Necessary Condition	A sub-condition feeding a CSF. The lower layer of a Goal Tree.	Entity type `necessaryCondition`.
AND group	A set of causes that are individually necessary but only jointly sufficient. Rendered as a junctor circle in TP Studio.	The `andGroupId` field on edges; JunctorOverlay renders it.
Sufficiency vs. necessity	A cause is *sufficient* if it alone produces the effect; *necessary* if the effect can't happen without it.	Edge `kind: 'sufficiency'` vs. `'necessity'`.
CRT — Current Reality Tree	The "Why is it this way?" diagram. Bottom-up causality.	Diagram type `crt`.
FRT — Future Reality Tree	The "What would it look like solved?" diagram. Same causal model, different starting node — an injection instead of a root cause.	Diagram type `frt`.
PRT — Prerequisite Tree	"What's in our way?" The obstacles between current and future, paired with intermediate objectives.	Diagram type `prt`.

Term	Meaning	TP Studio analogue
TT — Transition Tree	"How do we get there?" Action / precondition / outcome triples.	Diagram type `tt`.
S&T — Strategy & Tactics Tree	A deployment decomposition; each node is a 5-facet card (Necessary Assumption, Strategy, Parallel Assumption, Tactic, Sufficiency Assumption).	Diagram type `st`.
Verbalisation	Reading a diagram aloud, edge by edge, in natural language. The discipline that catches what scanning silently misses.	The Verbalisation Strip; the walkthrough overlay; the reasoning narrative export.
NBR — Negative Branch Reservation	The "yes, but…" analysis: an injection traced forward to the unintended UDE it might produce, so you can adopt, modify, or reject the change.	Diagram type `nbr`; also a group preset inside an FRT.
Browse Lock	A read-only mode you turn on before sharing or demoing the doc to prevent accidental edits.	⋮ overflow → *Lock for browsing*; the `browseLocked` flag.

Why a tool

You can do a CRT on paper. You can do an EC on a whiteboard. Practitioners have done so for decades. The reason a tool helps is *iteration*.

A CRT done well is not the diagram you draw on Monday. It is the diagram you have on Friday — after you've moved entities around five times, deleted three you thought were causes but turned out to be re-statements of the UDE, found the cloud that the conflict was hiding behind, added the assumption you didn't realize you were making, run the verbaliser and caught the gap, captured a revision, branched to explore an alternative, compared the two side by side, and came back to the original.

A tool — any tool — collapses the marginal cost of those revisions. Whiteboards have erasers, but erasers don't preserve the prior state. PowerPoint preserves state but punishes restructuring. A purpose-built TOC canvas lets you treat the diagram as something *you're allowed to be wrong about*, repeatedly, cheaply.

TP Studio is one such canvas. Where comparable practitioner guides have assumed Flying Logic, this book assumes TP Studio. Most of what follows would transpose to another canvas with light edits. What is specific to TP Studio is mostly *which palette command you press*, not *what you're doing when you press it*.

What's next

The next chapter (Your first canvas) is the 30-minute tool tour. After that, Chapter 3 — Reading a diagram covers the notation: causality conventions, edge polarity, AND / OR / XOR, the CLR. Then Part 2 opens with the Current Reality Tree, which is where the real work begins.

🔁 **Chain to next:** orient yourself to the surface, then learn to read what you'll later draw, then start drawing.

→ Continue to Chapter 2 — Your first canvas

Chapter 2 — Your first canvas

> *30-minute hands-on. You're going to open TP Studio, create an entity, connect two of them, and inspect the result. No method content here — pure orientation to the surface so the rest of the book can reference TopBar buttons and palette commands by name without you stopping to look them up.*

Opening the application

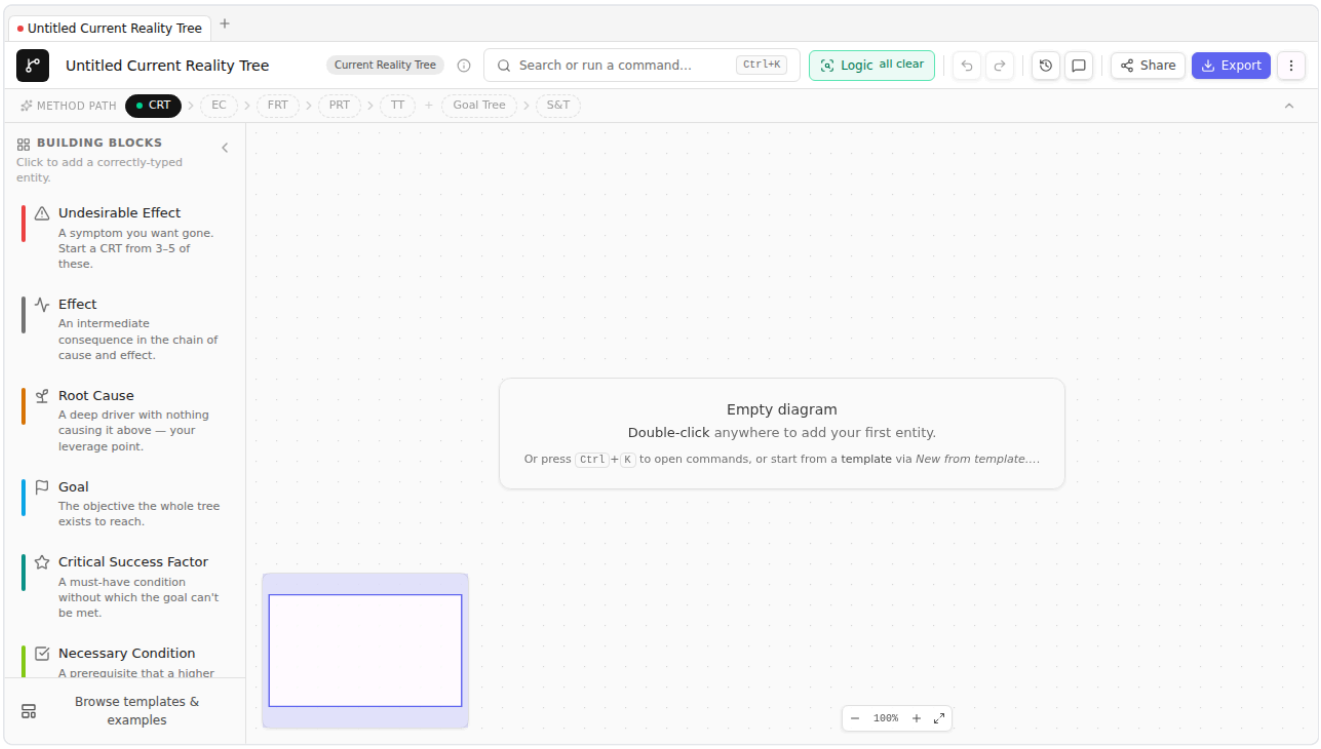
TP Studio runs in a browser tab. Three ways in:

- **The hosted PWA** at https://tp-studio.struktureretsundfornuft.dk/. Works offline after first visit; nothing leaves your machine.
- **A local dev server** if you cloned the repo: `pnpm dev`, then open the URL it prints (usually `http://localhost:5173`).
- **A local preview** of the built bundle: `pnpm preview` after `pnpm build`.

Pick the hosted PWA if you're a reader rather than a contributor. The app is identical either way.

Updates. When a new version of TP Studio ships, a small "New version available — Refresh now" toast appears at the bottom of the canvas; click it to apply. The flow is explicit on purpose: silent reloads while you're mid-edit would be hostile. If you want to check on demand (e.g. after a known release), `Cmd/Ctrl+K → Check for updates` forces the service worker to look — it tells you either *"You're on the latest version of TP Studio"* (green) or surfaces the refresh prompt for a new build (info).

TP Studio opens on the **Start** page — a workspace that names a problem for you, lists your trees, and offers templates (covered in The Start page below). Building or opening a tree from there drops you onto the canvas. A brand-new tree is an empty canvas with a centered hint:



> ***Empty diagram*** *Double-click anywhere to add your first entity.*

Your work auto-saves to this browser on every change. Closing the tab keeps the tree in the library (the Start page's "All trees"); reopening the app picks up where you left off. No sign-in, no cloud, no upload — your diagrams live in this browser's `localStorage` and nowhere else. (Sharing with others is a separate step covered in Chapter 16.)

What's on screen

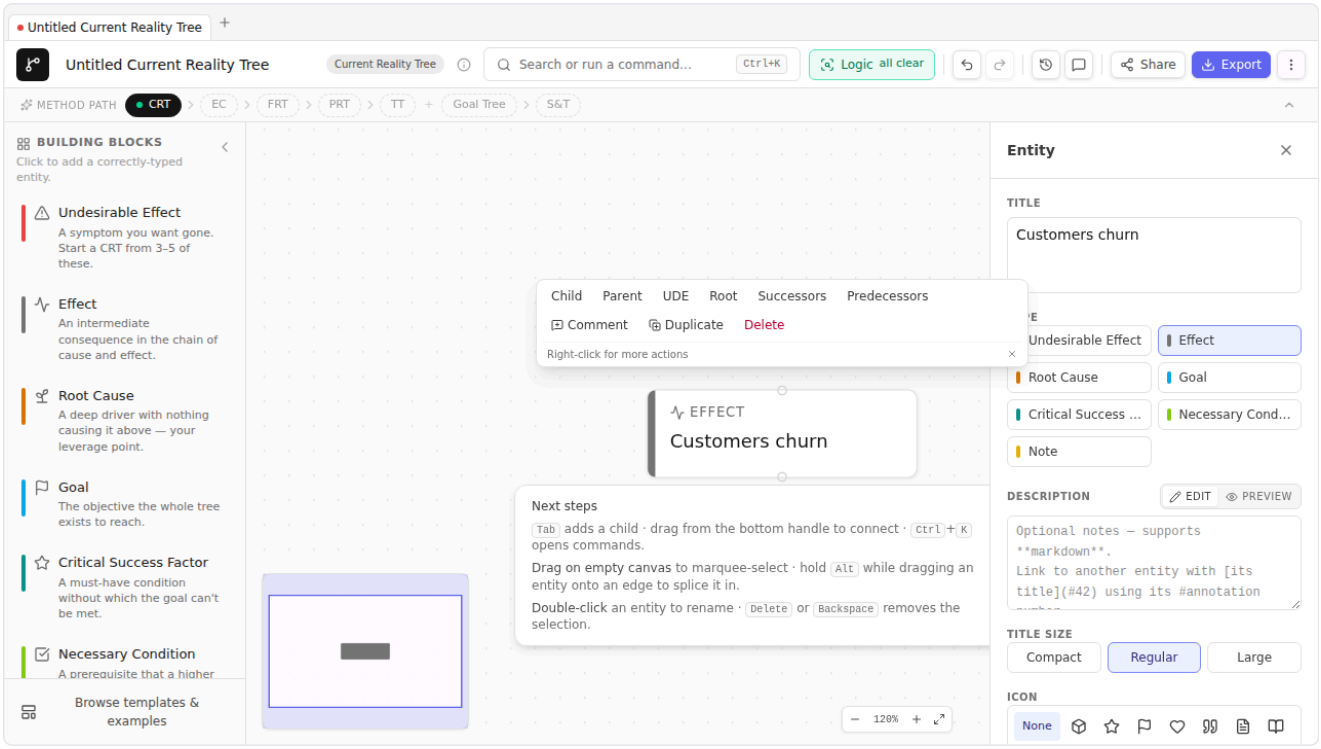
Element	Where	What it does
Home / logo	Top-left	The TP Studio mark. Opens the **Start** workspace — your trees, the templates, and the problem-led "build a CRT" hero (see below).
Title + type badge	Top-left	Click the title to rename. A `CRT` / `FRT` / `EC` / etc. badge shows the diagram type; the small ⓘ icon opens the Document Inspector.
Command search	Top-center	Click it (or press `Cmd/Ctrl+K`) to open the command palette.
Building Blocks rail	Left edge	Type-led entity creation for the current diagram — click a block to drop that entity at the canvas centre. Collapsible.
Method path	Strip under the top bar	Where the active diagram sits in the TP sequence (CRT → EC → FRT → PRT → TT, plus the Goal / S&T branch), with a suggested next step. **Collapsible** — hide it with its chevron, reopen from the ⋮ menu.
Logic check chip	Top-right	Emerald **"all clear"** or amber **"N to review"** — opens the CLR (Categories of Legitimate Reservation) panel.
Undo / Redo · History · Comments	Top-right	Step through edits; the Revision panel; review comments.
Share · Export	Top-right	Copy a read-only share link; open the unified Export picker.
Overflow (⋮)	Top-right	Theme, Browse Lock, Help, the layout-mode toggle, and **Show / Hide method path**.
Canvas	Center	The infinite dot-grid where your diagram lives. Pan with middle-click drag or two-finger scroll; zoom with the wheel or `+` / `-`.
Zoom controls	Bottom	Zoom in, zoom out, fit-to-view.
Inspector	Right panel	Slides in when you select an entity or edge — title, type, description, attributes, and warnings. Shares the dock with the Logic-check panel.
Toaster	Bottom-center	Brief confirmations: "Saved", "Loaded example CRT", "3 open CLR concerns".

Creating your first entity

Double-click the empty canvas anywhere. A blank node appears with the title field already focused. Type a short phrase — a noun-phrase describing something true about your system — and press Enter.

For this walk-through, type: **Customers churn**

Press Enter. The entity commits. You should see this:



That's an entity. By default it's an `effect` type — neutral grey stripe. The type tells you what role this node plays in the causal model; we'll get into types in Chapter 4.

Click the entity to select it. The Inspector slides in from the right with everything about this entity laid out: title, type grid, description, the rest. Try changing the type to `Undesirable Effect` by clicking the red-striped tile in the Inspector's Type grid. Notice the entity's stripe colour change on the canvas.

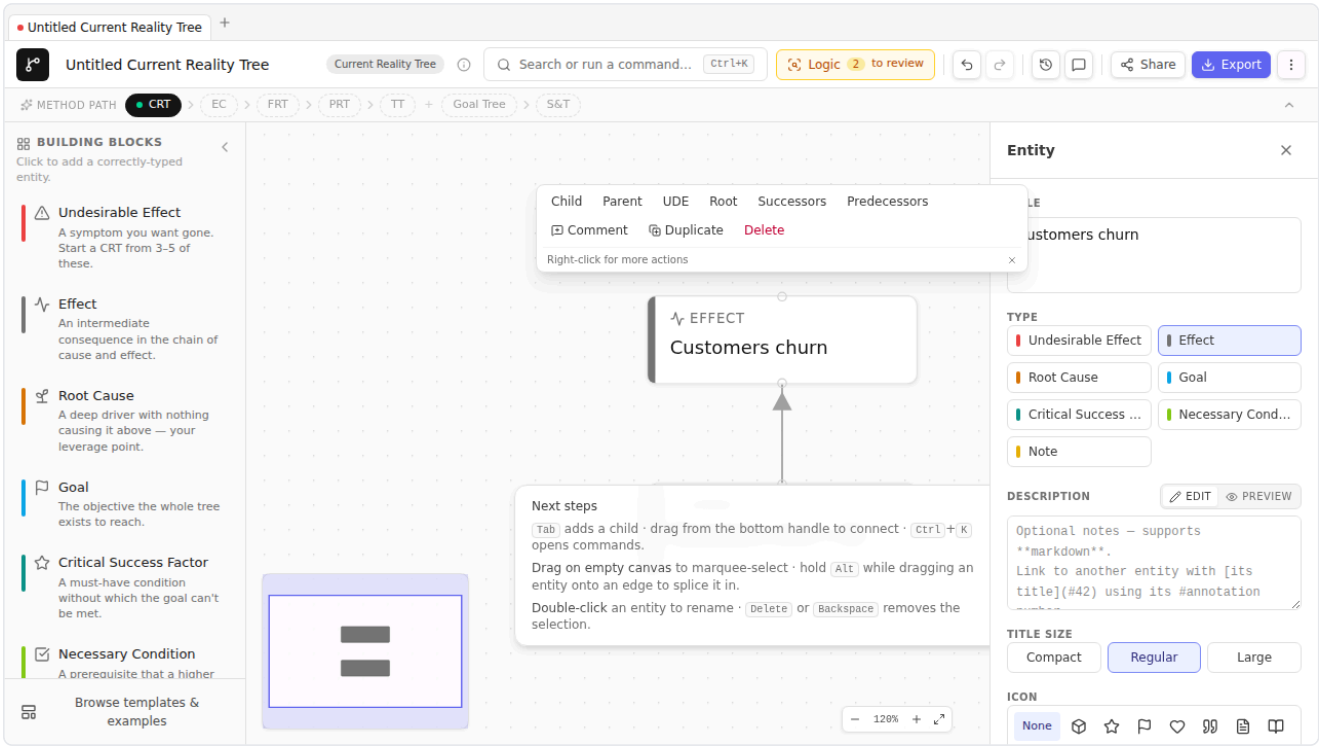
Connecting two entities

Double-click the canvas somewhere below the first entity. Type **Resolution time exceeds 8h** and press Enter. You now have two entities, side by side.

To connect them: hover your mouse near the bottom of an entity — small handle dots appear on the top and bottom edges. Click and drag from the bottom handle of "Resolution time exceeds 8h" up onto "Customers churn". Release. An arrow appears.

Or — faster — select "Resolution time exceeds 8h" by clicking it, then `Alt+click` "Customers churn". TP Studio interprets Alt-click as "connect from current selection to clicked node".

Your canvas now looks like this:



Read the arrow aloud: *"Resolution time exceeds 8h" causes "Customers churn".* That's the CRT reading convention — bottom-up, cause to effect, `because` linking the upper to the lower. (FRT, EC, PRT, and TT each have their own conventions; we cover them in Chapter 3.)

The command palette

`Cmd+K` (Mac) or `Ctrl+K` (Windows / Linux) opens the command palette — the canonical way to do anything in TP Studio that isn't a direct canvas gesture. Try it now. You'll see a search box and a list of commands grouped by category: File, Edit, View, Review, Export, Help. Your five most-recent commands pin to the top in a Recent group.

Type **Help** to filter. The top result is "Show keyboard shortcuts". Press Enter to open the Help dialog — that's the reference for every shortcut the application exposes.

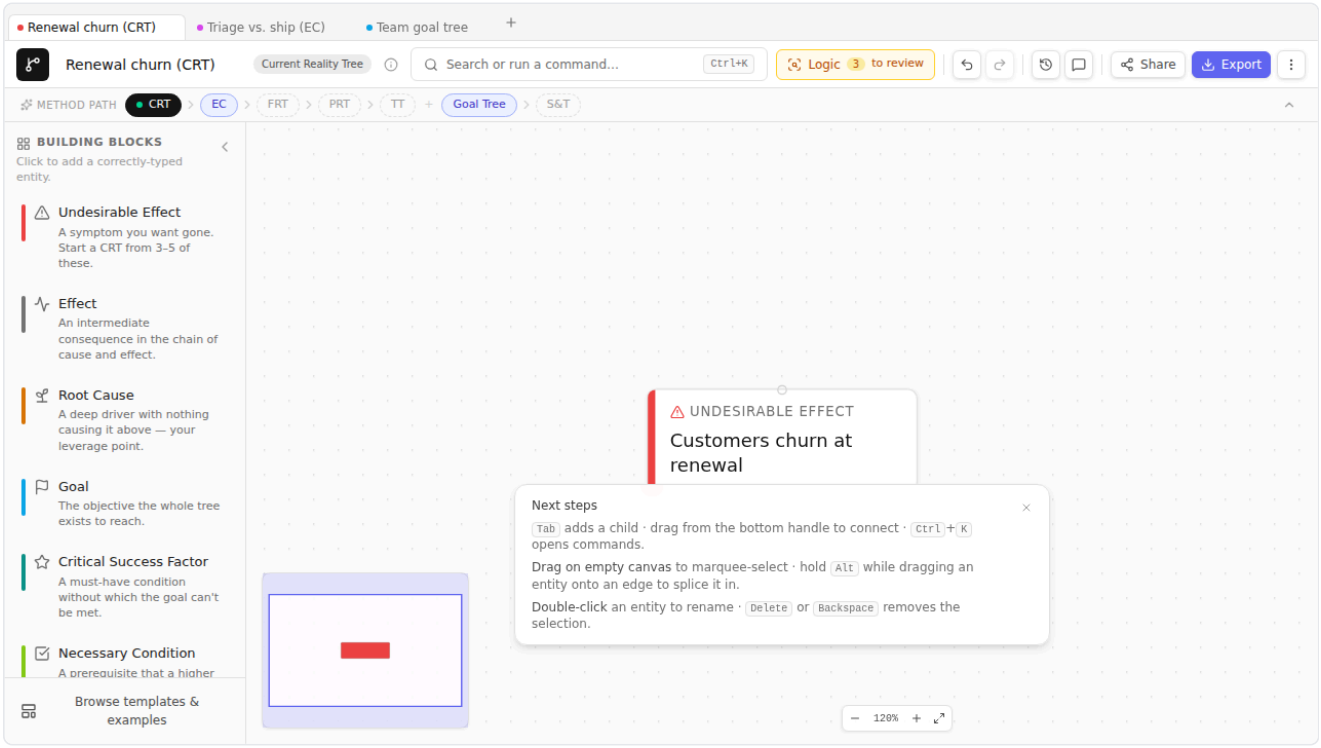
A few commands worth memorizing today:

Type into palette	What it does
`New diagram`	Open the picker for fresh CRT / FRT / PRT / TT / EC / Goal Tree / S&T / NBR / Freeform docs.
`Load example`	Open the picker for canned example docs in every diagram type.
`Browse templates`	Open the unified Templates library (~69 curated starters across every diagram type; `New from template` opens the same dialog).
`Export`	Open the unified Export Picker (PNG / SVG / JPEG / PDF / PPTX / Markdown / OPML / DOT / Mermaid / VGL / Flying Logic XML / share link).
`Capture snapshot`	Save a revision; the History panel will then let you compare or restore.
`Show keyboard shortcuts`	The full key reference.

The read-only **share link** lives on the top-bar **Share** button (and as a card in the Export picker) — it generates a URL that encodes the entire doc and loads it elsewhere in read-only mode.

Working with multiple documents

TP Studio keeps several documents open at once, each in its own **tab** along the top of the canvas. Click a tab to switch; click the **+** to open a fresh CRT; hover a tab and click **✕** to close it; drag a tab to reorder. Closing the last tab leaves a new blank one — there is never zero.



Each tab is independent: its own undo history, autosave, and share link. Reloading the browser restores *every* open tab, not just the active one.

Opening a document — an import, a pattern, a template, an example, a shared link, or an Evaporating Cloud spawned from a conflict — opens it in a *new* tab by default, leaving your current work in place. Prefer the old "replace what's open" behaviour? Turn off **Settings** → **Behavior** → **"Open documents in new tabs"** (Appendix D).

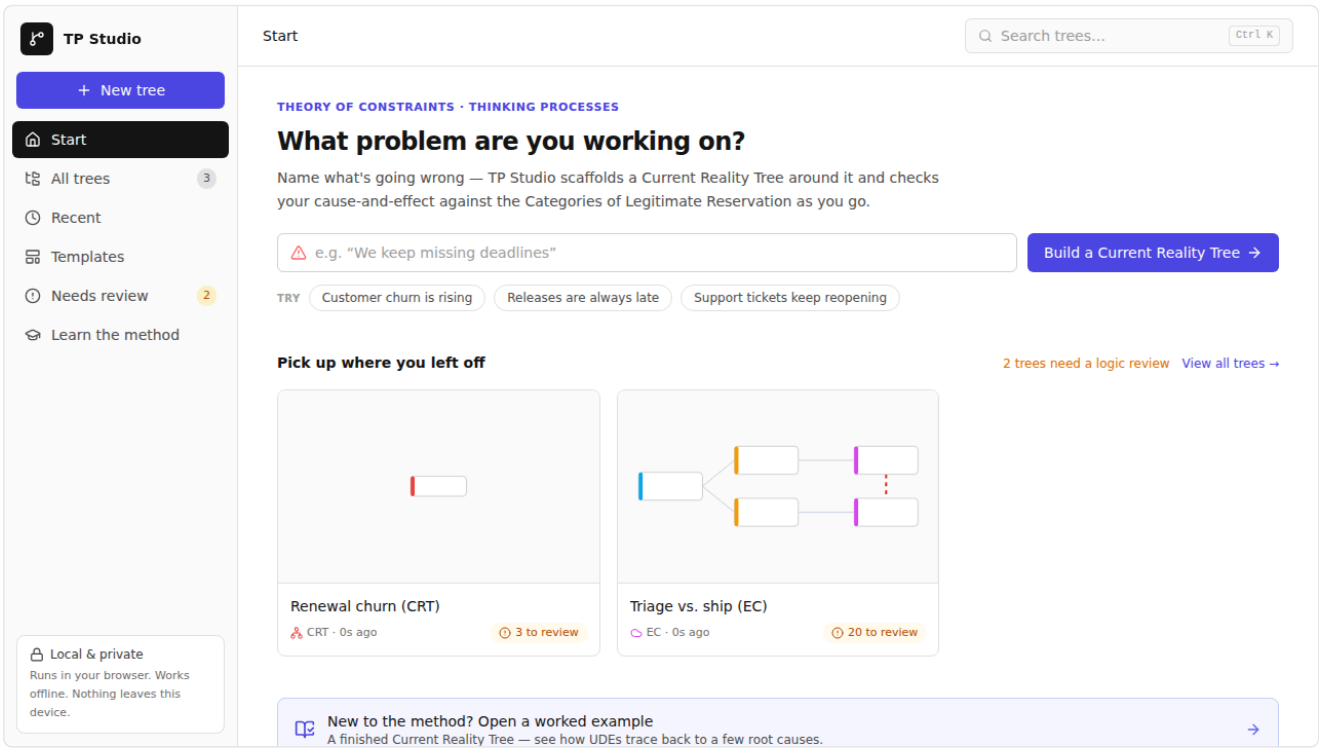
The palette (`Cmd/Ctrl+K`) carries **New / Duplicate / Close / Next / Previous tab** and **Forget closed documents** (reclaims storage from closed docs). Installed as an app, the native `Cmd/Ctrl+T` / `+W` / `+1` – `9` keys work too (Appendix B).

The Start page — your workspace home

TP Studio **opens on the Start page**: a full-screen workspace that sits in front of the editor, with a persistent left sidebar whose nav drives the main area. The **logo** (top-left) returns you to it any time.

- **Start** — a problem-led hero. Type what's going wrong (*"We keep missing deadlines"*) and **Build a Current Reality Tree** mints a fresh CRT with that statement as its first UDE — no blank canvas. Example chips do the same in one click; a worked-example callout opens a finished CRT to learn from; and a strip of templates sits beneath. Once you have work in progress, a **"Pick up where you left off"** row shows your most-recent trees with their logic status.

- **All trees / Recent** — every tree you've made as a card (a mini preview + title + type + when you last edited it) or a compact list. **Closing a tab keeps the tree here** — "All trees" is a library of every tree, not just the open tabs; hover a card and click the trash icon to delete one for good. Each card carries a **Logic pill**: emerald *"Logic clear"* or amber *"N to review"*, reading the exact same Categories-of-Legitimate-Reservation check the editor's Logic chip uses — so a card can never disagree with the canvas.
- **Templates** — the full template library, grouped by diagram type (Goal Trees, Evaporating Clouds, CRTs, …). Click a card to load it into a new tab.
- **Needs review** — just the trees with at least one open reservation. This is the CLR as triage: open the workspace, see which trees still have logic to resolve, and click straight in.
- **Learn the method** — the User Guide, the keyboard reference, and this book.



Clicking any tree card, template, or the **Build** button drops you straight into the editor on that document; **New tree** (top of the sidebar) opens the diagram-type picker. Trees stay in the library until you delete them — `Cmd/Ctrl+K → Forget closed documents` clears the closed ones in bulk.

Saving, exporting, sharing — the one-paragraph version

You don't save. TP Studio saves continuously to localStorage. Close the tab; reopen; your work is there.

You export by opening the Export picker (`Cmd+K → Export…` , or the top-bar **Export** button). The picker groups everything into five categories: **Images** (PNG / JPEG / SVG / copy to clipboard), **Documents** (PDF / print / PPTX / standalone HTML viewer / EC workshop sheet), **Data** (JSON / CSV / OPML / DOT / Mermaid / VGL / Flying Logic), **Annotations & reasoning** (Markdown narratives and outlines), and **Share**.

You share by either:

- Exporting the standalone HTML viewer (one file, no network, open by double-clicking on any machine); or
- Generating a read-only share link (the top-bar **Share** button) — a URL that contains the whole doc encoded into its fragment.

Sharing is fully covered in Chapter 16.

Try it

Five minutes. No reading.

1. Create three entities with double-click. Give them placeholder titles ("A", "B", "C" is fine).
2. Connect A → B → C using Alt+click.
3. Open the Inspector for B and change its type to `Root Cause`. Watch the stripe colour change.
4. Press `Cmd+K → Capture snapshot`. Name the revision "After type change".
5. Delete C. Notice the toast at the bottom.
6. Press `Cmd+Z` to undo. C comes back.
7. Open the History panel (the TopBar history button; it folds into the ⋮ overflow on narrow windows). You'll see your "After type change" snapshot.

You now know more about the surface than you can remember reading. That's the point of the chapter.

Starting from a real problem (not a blank canvas)

The hardest moment in any analysis is the empty canvas with a vague problem behind it. You don't have to start by drawing. Match your starting state to the on-ramp:

You have…	Start with	How
A problem you can name in a sentence	**The Start hero**	Click the logo, type the problem, **Build a Current Reality Tree**. Your sentence becomes the tree's first UDE — you're on a populated canvas immediately.
A vague mess, no idea where to begin	**Rapid 3-cloud diagnosis**	`Cmd+K → Rapid 3-cloud diagnosis…`. Name three symptoms and the tug-of-war behind each; it consolidates them into one Core Cloud to work from.
A brain-dump, meeting notes, a bulleted list	**Quick Capture**	Press `E` (outside a text field). Paste the indented list; each line becomes an entity, indentation becomes causal nesting.
A spreadsheet of items	**CSV import**	`Cmd+K → Import… → Entities CSV`. A `title,type,parent_title` header maps rows to entities and edges.
A sense that "someone's drawn this shape before"	**Templates**	`Cmd/Ctrl+K → Browse templates…` (or `New from template…` — same dialog). One unified library: system archetypes, domain CRTs, the classic clouds, and per-type starters. Load one, edit it into your situation.
A problem you can *describe* but not draw	**The AI skill**	Describe it to Claude via the `tp-studio-import` skill — "*a CRT for why onboarding churns*" — and import the `.json` it produces (Chapter 16). Treat it as a first draft to scrutinise, not an answer.

All six drop you onto a populated canvas with *something* to react to — and reacting is far easier than creating from nothing. From there it's the same loop: read it aloud, scrutinise the arrows, restructure, repeat.

Where this lives in the rest of the book

- Diagram-type-specific instructions live in the relevant Part 2 chapter (CRT in 4, EC in 5, etc.).
- Notation conventions live in Chapter 3.

- The CLR validators are Chapter 13.
- Revisions and side-by-side compare are Chapter 14.
- Exports, share links, and prints are Chapter 16.
- Every keyboard shortcut is in Appendix B.
- Every Settings toggle is in Appendix D.

🔁 **Chain to next:** you can drive the surface. Next you need to be able to *read* what you draw on it.

→ Continue to Chapter 3 — Reading a diagram

Chapter 3 — Reading a diagram

> *The notation chapter. Nothing here is original to TOC — these are the conventions Goldratt's tradition has settled on over forty years. The chapter is short because it's reference-shaped: come back to it whenever a later chapter says something like "necessity edge" or "AND junctor" and you need a quick refresher.*

Causality flows up (mostly)

In Current Reality Trees and Future Reality Trees, **causality flows from bottom to top**. Causes sit below; effects sit above. An arrow points *from* the cause *to* the effect. Read aloud: *"Because [cause], [effect]."* Or equivalently: *"[Cause] therefore [effect]."*

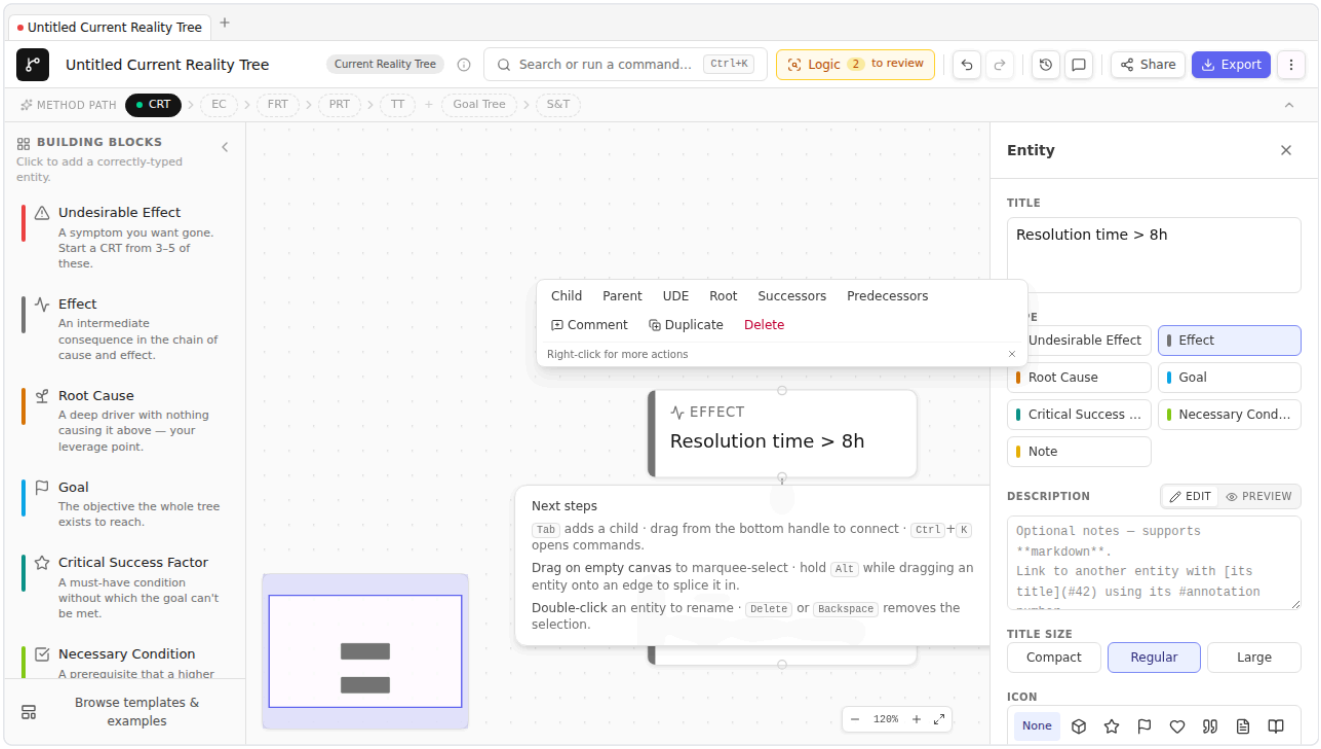
In Prerequisite Trees, Transition Trees, and Evaporating Clouds, the orientation is different — covered chapter-by-chapter. The point worth knowing now: **TP Studio respects per-diagram orientation conventions automatically.** When you load a CRT, dagre lays it out bottom-up. When you load a PRT, dagre lays it top-down. When you load an EC, you get a hand-positioned 5-box layout. Don't fight the convention; the convention is what makes the diagram *readable* to other practitioners.

Causality reading mode

The most common confusion when first encountering a CRT is "do I read up or down?" Goldratt's tradition says read up: *"because A, B; because B, C."* Some practitioners prefer the dual: *"A, therefore B; B, therefore C."* Both are valid.

TP Studio lets you pick a global default in **Settings → Display → Causality reading**:

Mode	What it does
`none`	No fallback label on edges. Best for clean exports.
`because`	Renders a muted italic "because" on each unlabeled edge. Reads bottom-up.
`therefore`	Renders a muted italic "therefore" on each unlabeled edge. Reads top-down.
`auto` *(default)*	Picks per-diagram: `because` for CRT / FRT / TT; `in order to` for PRT / EC; nothing for freeform / S&T.



Per-edge labels always override the global fallback. If you've explicitly labeled an edge — say, "because the SLA target is 8h" — the fallback word disappears and your label renders instead. Aggregated edges (those representing several edges collapsed across a group boundary) skip the fallback too; their `×N` count badge is the more informative thing.

Sufficiency vs. necessity

The most consequential distinction in TOC notation, and the one most easily fudged.

Sufficient cause: the cause, by itself, produces the effect. *"Because A, B"* — A alone is enough. CRT and FRT edges are sufficient causes.

Necessary condition: the effect *requires* the cause; without the cause, the effect can't happen — but the cause alone is not enough. *"In order to obtain B, we must have A"* — A is required, but other things might also be required. EC and PRT edges are necessity edges.

TP Studio's schema makes the distinction explicit. Each edge has `kind: 'sufficiency' | 'necessity'`, set per-diagram-type by default. You can override per-edge in the Edge Inspector when you have a mixed-causality diagram.

The verbaliser uses this field to choose between "because" / "therefore" wording and "in order to" / "we must" wording. The CLR validators use it too: a *Predicted Effect Existence* check fires only on sufficiency claims, because "if A is true, B should also be true *somewhere we can see*" is a sufficient-cause prediction.

Edge polarity

Classical Goldratt CRT notation was binary: an arrow either exists or it doesn't. Practitioner tools, Flying Logic most prominently, extended the notation through the 1990s and 2000s to support **polarity** — does this cause *increase* or *decrease* the effect? TP Studio adopts that extended notation for compatibility with the practitioner tradition. Three values, plus a default:

Polarity	Visual	Meaning
`positive` *(default)*	Plain arrow, no badge	"More of A → more of B." The classic sufficient-cause arrow.
`negative`	Arrow with a rose `−` badge	"More of A → less of B." A is a *reducer*.
`zero`	Arrow with a neutral `∅` badge	"A is flagged as non-influential here." Common in iterated FRTs where a previously-suspected cause has been ruled out.

Polarity is set via the Edge Inspector's Polarity picker or by cycling on the selected edge with the `cycle-edge-polarity` palette command. The cycle order is default → positive → negative → zero → default.

Polarity matters mostly in FRTs and S&T trees, where the question "will this injection actually move the needle?" hinges on whether the chain is all-positive (good), mixed (worth thinking), or contains a negative downstream of your lever (problem).

AND / OR / XOR junctors

A causal arrow on its own claims *sufficiency*: the cause produces the effect. But many real causal patterns aren't single-arrow — they involve *combinations*. Goldratt's original CRT included AND-grouping; the explicit OR and XOR junctors come from the same practitioner-tool tradition that introduced polarity (above), where they were given visual conventions Flying Logic popularized. TP Studio adopts those conventions:

Junctor	Visual	Logical meaning
AND	Violet circle labeled `AND`	All inbound causes must be present for the effect to occur. The set is jointly sufficient; no individual is.
OR	Indigo circle labeled `OR`	Any one of the inbound causes is sufficient. The set is alternative.
XOR	Rose circle labeled `XOR`	Exactly one of the inbound causes occurs; mutually exclusive alternatives.

To group edges into a junctor: select the edges (use shift-click for multi-select), then `Cmd+K → Group selected edges as AND` (or `OR` / `XOR`). The selected edges are rewired through a single junctor circle just below the target node, with one short outgoing arrow from junctor up into the target.

Cross-kind exclusivity: an edge belongs to at most one junctor kind. If you group an AND-grouped edge into an OR, the AND group dissolves first.

You'll mostly use AND. It's the conjunctive that makes a CRT honest: if "Customers churn" requires BOTH "Resolution time > 8h" AND "Onboarding is poorly documented", saying "either causes churn" is overstating the strength of each. The AND junctor forces you to be honest about which combinations are sufficient.

Back-edges

Sometimes a diagram has a real cycle. Customers churning *causes* lower retention metrics, which *cause* leadership pressure on the support team, which *causes* deferred refactors, which *causes* the original "resolution time exceeds 8h", which *causes* churn. Round and round.

TP Studio treats a cycle as structure to acknowledge, not a bug to flag. Cycles are **auto-detected**: the loop-closing edge — the one running against the diagram's flow — renders as a **back-edge** automatically, with a dashed stroke and a small `↻` glyph mid-edge. There is no cycle warning to suppress; the rendering *is* the acknowledgement. What the validators do watch on a loop is its dynamics: the **loop-polarity** check reads the loop as Reinforcing (R) or Balancing (B) and badges the back-edge accordingly, and **reinforcing-no-delay** nudges you when a reinforcing loop carries no delay marker (real feedback almost always lags).

You can also tag an edge as a back-edge explicitly — to choose *which* edge closes the loop, or to name the loop: select the edge, Inspector → Back-edge toggle, or right-click → "Tag as back-edge".

Most diagrams don't need back-edges. Use them when the structure is genuinely a positive or negative *reinforcing loop* (Senge's term) — a feedback structure you want to study, not a CRT cycle you accidentally drew.

Mutex (EC-only)

In an Evaporating Cloud, the two "Want" entities (D and D′) are *in conflict*. The whole point of the diagram is that you've identified a tension between two real, legitimate wants — they pull you in opposite directions.

TP Studio represents this conflict with a **mutex edge** between D and D′: rendered red with a lightning-bolt (⚡) glyph in the middle. It's the only edge in the diagram that's bidirectional in meaning ("they conflict" rather than "A causes B").

The `ec-missing-conflict` validator fires on any EC document until at least one want↔want edge is flagged as `isMutualExclusion`. Tag it via the Edge Inspector's **Mutual exclusion (EC)** checkbox (it appears when both endpoints are Wants).

Locus

A second extension Goldratt added: each entity carries a **Locus** flag (in earlier versions of TP Studio this was labelled "Span of Control") indicating who controls it.

Locus	Visual	Meaning
`control`	Emerald `C` pill	Inside your control. You can directly change this.
`influence`	Amber `I` pill	Outside your direct control, but you can plausibly influence. Vendors, neighboring teams, customers via product changes.
`external`	Neutral `E` pill	Outside your control AND your influence. Macro conditions, regulation, market forces.

Why it matters: **the constraint you want to find lives in your control or influence zone**, not in the external zone. The `external-root-cause` CLR validator fires on root causes flagged `external` — the diagram tells you you've found a "cause" that you can't actually do anything about, which usually means you've stopped one step short of the real root cause.

Set via Entity Inspector → Locus, or right-click → Set locus.

Ownership and validation

The Inspector also carries an **Owner** field — a free-form text input naming whoever's accountable for the entity. Decision owner on a UDE, action assignee on a TT action, validation owner on an assumption — whatever role makes sense for the entity's place in the diagram. Below the field is a **Mark validated** button that stamps a time-stamp into `entity.lastValidatedAt`; on subsequent visits the timestamp reads back as "Last validated YYYY-MM-DD by <owner>" so an audit trail accumulates across re-validations.

Two consumers downstream:

1. The **Risk Register (CSV)** export (Chapter 16) reads the field directly — the `owner` column populates from `entity.owner`.
2. Future collaboration features will use the same field as the human-readable name for the "who" of accountability without needing a formal user model.

The legacy `attributes.owner` key still works for older docs (the risk-register export checks both, preferring the dedicated field), but new docs should use the typed Owner field.

Evidence — making provenance explicit

Beneath the Owner block is an **Evidence** list — a structured place to record *what the entity is standing on*. Each row carries:

- A **description** — the actual citation, observation, measurement, or claim. Free text.
- A **source** pill — cycles through `Observed → Stakeholder → Metric → Policy → Assumption`. The taxonomy is intentionally narrow:
  - **Observed** — first-hand observation ("I saw the queue grow during demo days")
  - **Stakeholder** — an assertion from someone with stake ("the CFO says renewals will drop 12%")
  - **Metric** — a numeric measurement ("p95 latency = 740ms last week")
  - **Policy** — a documented rule or constraint ("PCI requires segregated networks")
  - **Assumption** — explicitly an unproven belief, kept honest by the label rather than masquerading as fact
- A **strength** pill — `Weak / Moderate / Strong`. Your qualitative judgement; nothing derives it for you.
- An optional **URL** — citation link, dashboard URL, document reference.
- A per-row **Mark validated** button — stamps the current timestamp and credits the entity's Owner as the validator. Distinct from the entity-level validation: the entity audit-trail tracks "is this entity still true overall"; the per-evidence trail tracks "is THIS specific citation still standing."

Use the list to make provenance explicit at workshop time. A UDE backed by `[strong/metric] p95 = 740ms` reads very differently from one backed by `[weak/assumption] customers probably care about latency`, and the difference often surfaces during the trim step of an FRT — the moderate-evidence claims are the ones to challenge first.

Evidence items survive JSON export / share-link reload, and feed the new `evidence` column of the **Risk Register (CSV)** export — rendered as semicolon-joined `[strength/source] description (url)` entries, so a register reader gets the qualitative signal at a glance without losing the original detail.

Entity state and what-if speculation

Reading a diagram is one thing; *interrogating* it is another. TP Studio lets you assign each entity a **state** — the answer to "is this true right now?"

State	Badge	Meaning
`true`	Emerald `T`	The entity holds. The cause is present; the effect has occurred.
`false`	Rose `F`	The entity does not hold.
`unknown` *(default)*	Neutral `?`	Undecided — no claim either way.
`disputed`	Amber `?`	Stakeholders disagree; recorded as contested rather than resolved.

Set a state from Entity Inspector → State, or by cycling the selected entity. A small state badge then rides the node's corner.

What makes states more than annotation is **propagation**. TP Studio folds known states downstream along the causal structure: a `true` cause whose junctor is satisfied lights its effect `true`; a `false` member of an AND junctor blocks the effect; OR/XOR fold by their own rules; negative-polarity edges flip the signal; zero-polarity and back-edges are skipped; cycles short-circuit rather than loop forever. Derived states render in the same badges but without the manual marker, so you can always tell what *you* asserted from what the diagram *concluded*.

What-if speculation is the live version of this. `Cmd+K → Speculate: what changes if…` opens a temporary overlay: flip any entity's state and watch propagation re-flow across the canvas in real time, with speculated badges drawn with a dashed ring so they're visibly provisional. A banner offers **Commit** (write the speculated states back as the real ones, in a single undo step) or **Revert** (`Esc`) — discard the overlay and restore reality. Nothing touches the saved document until you commit.

Use it to ask the questions a static diagram can't: "If this assumption turns out false, what downstream effects collapse?" "Which root cause, flipped to true, lights up the most of the tree?" Speculation turns the diagram from a picture into a model you can poke.

The CLR — a one-paragraph preview

Six "Categories of Legitimate Reservation" — the discipline checks Goldratt taught for evaluating someone else's causal claim. TP Studio surfaces them automatically as **warnings** in the Inspector's Warnings list. They are not errors; they are reservations a thoughtful colleague would raise if they were reading your diagram over your shoulder. We get into the CLR in depth in Chapter 13, but for now know that:

- Warnings fire on entities AND edges, depending on what the rule checks.
- Tiers escalate from `clarity` (the most pedagogical) through `existence` to `sufficiency` (the more structural).
- You can dismiss warnings explicitly when you've considered the reservation and decided it doesn't apply.
- The `Start CLR walkthrough` palette command iterates all open warnings one at a time with Resolve / Open-in-inspector actions.

Quick reference card

Print this and tape it next to your monitor:

```
CRT / FRT       — causality flows up.    "Because [below], [above]."
PRT             — IO → obstacle.          "To get [above], must overcome [below]."
TT              — action + precondition.  "Do [action] given [precondition] to obtain
[outcome]."
EC              — 5 boxes in fixed layout. Read aloud per Chapter 5.
Goal Tree       — Goal → CSFs → NCs.      Top-down decomposition.
S&T             — 5-facet cards.          Top-down strategic deployment.

Edge polarity:  default • positive • negative (−) • zero (⌀)
Junctors:       AND (violet) • OR (indigo) • XOR (rose)
Back-edge:      dashed + ↻ glyph (auto-detected; R/B badge shows loop polarity)
Mutex (EC):     red + ⚡ glyph
Locus:          C (emerald) • I (amber) • E (neutral)
Entity state:   T (true) • F (false) • ? (unknown/disputed) • dashed ring = speculated
```

🔁 **Chain to next:** you can read what's on the canvas. Now build one from scratch.

→ Continue to Chapter 4 — Current Reality Tree

Chapter 4 — Current Reality Tree

Why is this happening?

> 🎯 ***What this process is for*** *A Current Reality Tree (CRT) traces a system's painful symptoms — the Undesirable Effects — back to their underlying cause. It answers: "Why is the system producing this?" Done well, it identifies the* ***core driver*** *— the single root cause whose dismantling would eliminate most of the symptoms. That's the constraint.*

The premise

The CRT is the first Thinking Process you reach for, almost every time. It's the *diagnosis*. You don't fix anything by drawing a CRT — fixing happens later, in the Future Reality Tree and Transition Tree. But you can't fix what you can't see; the CRT is what you see *with*.

The premise that makes CRTs work is the one we set up in Chapter 1: every system's underperformance traces to one constraint. Most of what looks like multiple problems is one problem in different costumes. The CRT's job is to reveal that the multiple-symptom appearance is illusory and the underlying cause is singular.

When this premise holds, a CRT for a system with 7-10 visible UDEs typically resolves down to 2-4 root causes, with one of them feeding 70-80% of the UDEs by reach. That last one is the core driver. (If your CRT has 7 UDEs all tracing back to 7 independent root causes, you either drew it wrong or — rarely — you're looking at a genuinely young, small system that hasn't yet accumulated structural pathology. In the latter case, congratulations.)

The method, neutral of tool

1. **List the UDEs.** Talk to the people closest to the system. Ask: "What does this system do that you wish it didn't?" Aim for 5-12 noun-phrases, each describing a single observable effect. ("Customers churn." "Support cost per ticket is up 40%.") Avoid causes ("Because we don't have a triage rubric…") — those go in later.
2. **Pick one UDE and ask why.** For each cause-candidate, ask: "Is this present because of something else?" — and add that something else as a node below. Connect them with a sufficiency arrow upward.
3. **Repeat for every UDE.** Most causes will recur across UDEs — that's the structure you want. A cause that doesn't recur is suspicious; either it's specific to one UDE (which is rare in a real system) or you're stopping too shallow.
4. **Identify converging cause-chains.** When two UDEs trace back through different intermediate effects to the *same* root cause, you've found a structural pattern. Mark it. The convergence is the constraint becoming visible.
5. **Find the core driver.** The root cause with the highest *reach* — the most UDEs it ladders up to — is the candidate. TP Studio's `Find core driver(s)` palette command does this calculation explicitly.
6. **Verbalize.** Read each cause chain aloud, edge by edge. "Because A, B. Because B, C." If a sentence sounds wrong, the diagram is wrong. Fix the diagram.
7. **Stop when the root cause is in your control or influence.** If you bottom out at a cause you flag `external` (the geopolitical situation, the macroeconomy, the customer's mood), keep going one level — there's almost always a `control`-level cause beneath an `external` one. The system you're analyzing belongs to *you*, not the macroeconomy.

The worked example

We'll build a CRT for a fictional but realistic problem: **a B2B SaaS support team chronically firefighting**.

The setup: VP of Customer Success calls you in. "We're losing customers. Support is burnt out. Cost per ticket is up. We've tried hiring more agents and it didn't help. We've tried better tooling and it didn't help. What's actually going on?"

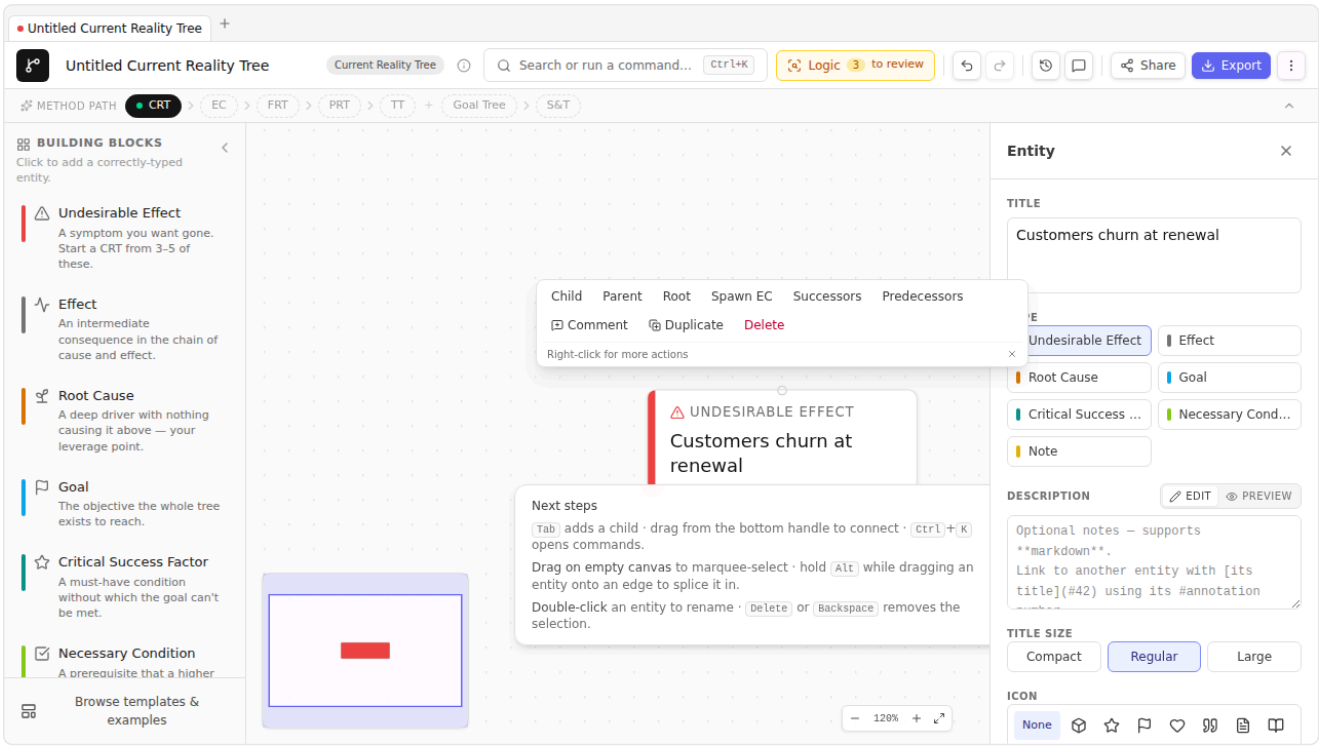
Three UDEs to start. You write them down in TP Studio.

> ***Scope before you draw.*** *Open a fresh CRT with an empty System Scope and TP Studio drops a one-time toast pointing you to the Document Inspector's* ***System Scope*** *section — Goldratt's Step 1 (name the system, its boundary, its measures)* before *the effects. Fill any answer and the nudge is satisfied for that document; dismiss it and it won't return. We jump straight to the UDEs to keep the example moving, but in real work, scope first.*

Step 1 — The first UDE

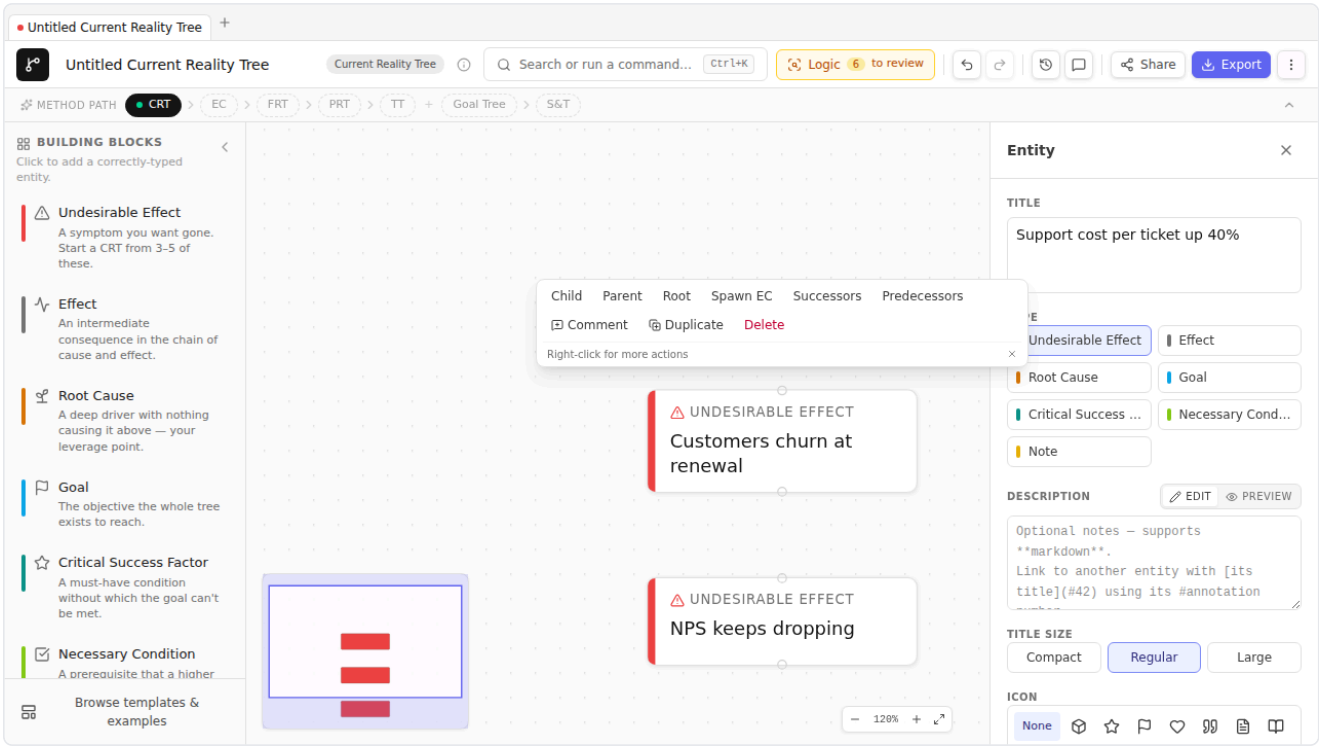
Open TP Studio. `Cmd+K → New diagram → Current Reality Tree`. Empty CRT canvas opens.

Double-click the canvas, type **Customers churn at renewal**, press Enter. Click the entity to select it. In the Inspector's Type grid, click **Undesirable Effect**. The stripe turns red.



Step 2 — The other UDEs

Two more double-clicks. Type **NPS keeps dropping** for one, **Support cost per ticket up 40%** for the other. Mark both as `Undesirable Effect` via the Inspector. Three red-striped entities side by side:



🛠 **How TP Studio helps:** The TopBar's TitleBadge will now say "CRT" — confirming the diagram-type-appropriate validators are active. The CLR walkthrough wizard (`Cmd+K → Start CLR walkthrough`) will already be flagging "UDEs without cause chains" as a clarity-tier warning. We'll get to those.

Step 3 — First cause-chain

Pick the most visceral UDE first. "Customers churn at renewal" — why? Because (you ask the team) deals close on the assumption of a 4-hour SLA but support routinely misses it. So: under "Customers churn at renewal", add a new entity titled **Customer SLA expectations are missed**. Connect upward.

Read it: *"Because the customer SLA expectations are missed, customers churn at renewal."* Sounds reasonable.

Why are SLA expectations missed? Because resolution time runs over 8 hours on most tickets. Add **Resolution time > 8h on most tickets**; connect.

Why? Two reasons, you discover after talking to agents. (1) The team has no shared triage rubric — every ticket gets the same level of investigation regardless of severity. (2) Senior agents context-switch constantly between L1 and L2 tickets, so neither gets done quickly.

Add both as causes. These are *jointly* sufficient — you need them both for the long resolution times. Group them as an AND. (Select both edges → `Cmd+K → Group as AND`.)

Step 4 — The second cause-chain

"NPS keeps dropping" — why? You ask. The biggest detractor theme is "I get a different answer every time I contact you." So: under "NPS keeps dropping", add **Customers get inconsistent answers**.

Why inconsistent? Because — again — no triage rubric. *Same* cause we already added under the other branch. So instead of duplicating it, connect "Customers get inconsistent answers" to the existing **No shared triage rubric** node. The arrow goes up; the node already lives over there.

This is the moment a CRT starts to *feel* like a structural diagnosis. The convergence is the constraint becoming visible.

Step 5 — The third cause-chain

"Support cost per ticket up 40%" — why? Two reasons emerge in conversation: agents redo prior work because there's no canonical answer to common questions; *and* senior agents are bottlenecked because everything escalates to them (which is the same context-switching problem from the first chain).

Add **Agents redo prior work for common questions**. Why? Because there's no consolidated answer base. Why no answer base? Because nobody's been protected from incoming tickets long enough to build one.

Add **Support lead has no protected drafting time**. Connect upward. Also: the senior-agent context-switching pulls in the same direction. Connect.

You should now have something resembling a tree with three UDEs at the top, converging through intermediate effects into a small number of root causes at the bottom. Click each terminal node and mark it `Root Cause` via the Inspector's Type grid.

Step 6 — Find the core driver

`Cmd+K → Find core driver(s)`. TP Studio computes UDE-reach for each root cause and selects the top candidate(s). The toast tells you the score: how many UDEs the highest-reach root cause ladders up to.

In our example: **Support lead has no protected drafting time** is reaching all three UDEs (directly or via the chain). That's the core driver. The constraint isn't "we need more agents" or "we need better tooling" — it's that nobody has been protected from incoming work long enough to build the system the team needs.

🛠 **How TP Studio helps:** Settings → Display → Show UDE-reach badge toggles an amber `→N UDEs` pill on each entity. With it on, the core driver visually announces itself in any CRT.

Step 7 — Verbalize

Click `Cmd+K → Start read-through` to open the walkthrough overlay. TP Studio iterates each structural edge in topological order. Read each sentence aloud.

"Because Support lead has no protected drafting time, agents redo prior work for common questions." Sounds right.

"Because Support lead has no protected drafting time, no shared triage rubric exists." Sounds right.

"Because No shared triage rubric AND Senior agents context-switch constantly, resolution time > 8h on most tickets." Sounds right.

"Because resolution time > 8h, customer SLA expectations are missed." Sounds right.

"Because customer SLA expectations are missed, customers churn at renewal." Sounds right.

Read each chain twice. The second time you'll catch the one that sounds *almost* right but isn't quite — a CLR tier-3 (cause-effect reversal) or tier-4 (predicted effect existence) violation. Fix it. Re-read.

Step 8 — Stop

How do you know the CRT is done?

- Every UDE traces to a root cause that's in your `control` or `influence` zone.

- The core driver is identified; its reach feels structurally honest (not "I made one node connect to everything"; really *connecting*).
- Read-aloud passes the gut check on every chain.
- The CLR walkthrough is empty (no open warnings) — or each open warning has been explicitly dismissed with a "considered and doesn't apply" note in the entity's description.

When all four are true, the CRT is ready for the next step — which is usually Chapter 5, where you ask why the core driver has persisted.

Feedback loops, the R/B badge, and system archetypes

A CRT is usually a directed acyclic graph — causes flow upward to effects and stop. But some systems bite back. A cause produces an effect that loops around and *amplifies the original cause*. When that happens, you have a feedback loop, and your CRT has a cycle. TP Studio detects the back-edge closing the cycle, classifies the loop, and tells you what kind of trouble you're in.

Building the loop — a "Fixes that Fail" example

Return to the support team scenario, and add one more cause chain that the team mentions almost in passing: "Whenever a major ticket causes a production outage, the on-call engineer restarts the affected service. That gets things working again in under ten minutes, so everyone moves on."

Add two new entities: **Production outage occurs** (mark it UDE) and **On-call engineer restarts service**. Connect **Production outage occurs → On-call engineer restarts service** (the immediate response), then connect **On-call engineer restarts service → Root fault goes uninvestigated** (the restart buys time but masks the underlying defect), and finally connect **Root fault goes uninvestigated → Production outage occurs** (the fault persists and triggers the next outage). That last arrow closes the cycle — it is the back-edge.

The moment you draw it, TP Studio decorates that edge with a small badge: **R**. That badge means *Reinforcing*. TP Studio derives it by multiplying the polarities of every edge around the loop — if the product is positive, the loop amplifies whatever is already happening. Here, each restart relieves the outage *and* prevents the fix, so outages accumulate over time. The R badge is the diagram telling you: this loop will not self-correct.

> ***R vs B at a glance.*** *A Reinforcing loop has no governor — it compounds. A Balancing loop has a target it steers toward; it self-corrects (though it may hunt, overshoot, or oscillate with a delay). Neither is inherently good or bad. A B loop can trap a system in a suboptimal equilibrium; an R loop can be the engine of growth or a death spiral. The badge just names what the structure does.*

Running the CLR check for the loop

Open the CLR walkthrough (`Cmd+K → Start CLR walkthrough`). If the loop is recognized, you'll see a **Loop polarity** entry in the clarity tier: *"Loop 'unnamed' is Reinforcing (R). Check whether this escalation is intentional."* That's the validator acknowledging the cycle and confirming its classification. If the loop spans a long chain, the walkthrough also flags any edge whose polarity you haven't explicitly set — because a mismarked polarity silently flips the R/B determination.

Naming the loop and noting its behaviour over time

Click the back-edge (the arrow from **Root fault goes uninvestigated** back to **Production outage occurs**). The Edge Inspector panel opens on the right. In the **Loop name** field, type `Restart spiral`. In the **Behaviour over time** note field, write: *"Outage frequency and severity escalate over time; each quick restart erodes the team's incentive to invest in a real fix."* These two fields travel with the back-edge — they become the loop's identity in any export or review.

Marking the delay

The fault doesn't cause the next outage instantly — there's typically a lag of days or weeks before the latent defect degrades enough to fire again. Select the edge from **Root fault goes uninvestigated → Production outage occurs** and toggle **Delayed** in the Inspector. TP Studio renders a `//` glyph on that edge, the conventional delay marker.

Before you mark the delay, the CLR walkthrough may have surfaced a warning: *"A reinforcing loop with no delay would escalate instantly — is a time lag missing?"* That warning goes quiet once a delay is marked. This matters practically: a reinforcing loop with a short delay looks like steady escalation; one with a long delay looks like isolated incidents with no obvious connection. The `//` glyph makes the lag visible to everyone who reads the diagram.

System-archetype patterns

What you just drew is one of five classical feedback patterns — *system archetypes* — that Peter Senge catalogued in *The Fifth Discipline*. TP Studio ships them as ready-to-load starters in the Templates library (`Cmd/Ctrl+K → Browse templates…`):

- **Fixes that Fail** — a symptomatic fix relieves pressure, masking the root cause, so the problem returns worse. *Counter-intuitive lesson: treat the root cause, not the symptom.*
- **Escalation** — two parties each respond to the other's actions by raising the stakes, amplifying the conflict. *Lesson: change the measure both sides are reacting to.*
- **Shifting the Burden** — a symptomatic solution crowds out investment in the fundamental solution, making the system dependent on the symptom-fix. *Lesson: invest in the fundamental solution before the dependency sets in.*
- **Eroding Goals** — when a gap between goal and performance is closed by lowering the goal rather than raising performance. *Lesson: hold the goal.*
- **Limits to Growth** — a reinforcing growth engine hits a limiting factor that slows and eventually stops growth. *Lesson: lift the limit, not the growth rate.*

Load "Fixes that Fail" as a starting point when your situation smells like the restart spiral above. The loaded archetype drops a pre-wired loop onto the canvas; replace its placeholder labels with your own entities, and the R/B badges and CLR checks carry over automatically.

The support team's restart spiral is a textbook Fixes that Fail: the restart is the symptomatic fix, the outage recurrence is the delayed consequence, and the real fix — a mandatory post-incident root-cause investigation protected from the next incoming ticket — is the fundamental solution that never gets funded because the restarts keep the pain just bearable enough to defer it. Sound familiar?

Sidebars

> **🛠 How TP Studio helps**
>
> - `Cmd+K → New diagram… → Current Reality Tree` to start a fresh CRT.
> - `Cmd+K → Load example… → Current Reality Tree` for a 6-entity reference doc to study before drawing your own.
> - **Inspector Type grid** with one-click switching between Effect / UDE / Root Cause. (Assumptions aren't a type — they're edge annotations, added from the Edge Inspector's Assumption Well or by pressing `A` on a selected edge.)
> - **AND grouping**: select multiple edges → `Cmd+K → Group selected edges as AND` (or right-click → Group as AND).
> - `Find core driver(s)` palette command — picks the highest-reach root cause(s).
> - **UDE-reach badge** (Settings → Display → "Show UDE-reach badge") visualizes reach per entity.
> - **Reverse-reach badge** shows the symmetric "root causes per UDE" count, useful for verifying that each UDE has a real cause chain rather than dangling.
> - **CLR walkthrough**: `Cmd+K → Start CLR walkthrough` iterates open warnings.
> - **Read-through overlay**: `Cmd+K → Start read-through` — the verbalisation discipline made gesture.

> **💡 Practitioner tips**
>
> - **Start with UDEs, not causes.** Resist the urge to write your hypothesized causes first. Let the causes emerge from "why does that happen?" questioning. A CRT built top-down from UDEs is more honest than one built bottom-up from preferred conclusions.
> - **Re-use existing intermediate nodes.** When a second cause chain wants to land at the same intermediate effect you already drew, connect to the existing node — don't duplicate. The convergence IS the diagnosis.
> - **Don't be afraid to mark an entity** `unspecified`**.** The Inspector has an "unspecified — fill in later" toggle. Use it when you know there's a step in a chain but can't yet articulate it; it suppresses the empty-title warning while keeping the structure visible.
> - **Verbalize early, not just at the end.** Read each new chain aloud as you add it. The error correction is cheaper if you catch the awkward sentence before you've built three more entities downstream of the awkward one.

> ⚠ ***Common mistakes***
>
> - ***Listing causes as UDEs.*** *"We have no triage rubric" is not a UDE — nobody feels that directly. "Resolution time exceeds 8h" is a UDE. The UDE layer is what customers / employees / the market would say is wrong, not what you the analyst would say is wrong.*
> - ***Bottoming out at an external cause.*** *"Macroeconomic conditions are tough" is not a root cause for your CRT. Almost always there's a* `control` *-level cause underneath ("we haven't refreshed our pricing model since the rate hike"). The* `external-root-cause` *warning flags this.*
> - ***Drawing a CRT without an AND group when one is needed.*** *If A and B are both required for C, drawing them as two single arrows to C is technically wrong — it claims each is sufficient. Group as AND.*
> - ***Treating the CRT as the deliverable.*** *The CRT is a diagnostic intermediate. The deliverable is usually the FRT / TT pair that follows. If you find yourself polishing a CRT for a stakeholder pack, you've stopped one diagram too early.*

> 🛑 ***When to stop***
>
> - *Every UDE has a path to a* `control` *or* `influence` *root cause.*
> - *The core driver is identified and its reach is honest.*
> - *The CLR walkthrough is empty or every open warning is intentionally dismissed.*
> - *You can read each chain aloud without rewording.*
> - *You're ready to ask the next question: why hasn't this been fixed already? (That's the EC in Chapter 5.)*

> ✏️ ***Now you try.*** *Pick a recurring frustration on your own team. Open a CRT (* `Cmd+K → New diagram… → Current Reality Tree` *), capture three UDEs you can actually observe, and build each down to a root cause with* `Shift+Tab` *. Run* `Cmd+K → Find core driver(s)` *— does the cause it picks match your gut? Read the tree aloud with* ***Start read-through****, then clear the CLR walkthrough. If your tree reads like a to-do list, you've drawn a plan, not a diagnosis — restart with effects, not actions.*

🔁 **Chain to next:** the CRT shows you *what* is wrong. The Evaporating Cloud shows you *why nobody has fixed it yet*.

→ Continue to Chapter 5 — Evaporating Cloud

Chapter 5 — Evaporating Cloud

Why are we stuck?

> 🎯 ***What this process is for*** *An Evaporating Cloud (EC) reveals the conflict that holds a chronic problem in place. It answers: "Why hasn't this been fixed already?" The diagram is a five-box pattern showing two legitimate but opposing wants, each rooted in a real need, both subordinated to a common goal. When you draw it well, the cloud "evaporates" — you find the false assumption that made the conflict feel necessary, and the resolution emerges with it.*

The premise

If a problem has persisted in a system populated by reasonable people, **it's persisted for a reason**. Most chronic organizational problems aren't "we haven't gotten around to fixing them yet" — they're "every time we try to fix them, we create a different problem instead, and that other problem feels worse, so we revert." The thing that makes you revert is the conflict.

The Evaporating Cloud forces you to write that conflict down. Not as "Engineering vs. Product" (organizational shorthand) or "we need to choose between speed and quality" (false dichotomy in disguise), but in the specific 5-box structure that lets you see *which assumption* is making both sides feel they must hold their position.

Once the false assumption is named, the cloud evaporates. The "conflict" was never between the wants; it was between two ways of getting to the same goal under a constraint that no longer applies or never applied in the first place.

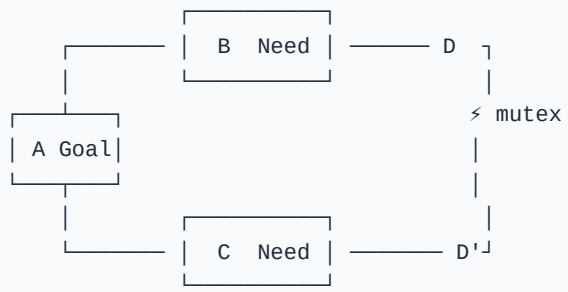
This is Goldratt at his most provocative. He claimed — and TOC practitioners since have largely confirmed — that almost every chronic conflict in an organization is *false*. The wants are real, the needs are real, the goal is shared; what's wrong is the unstated assumption that you must choose. The EC's job is to drag that assumption into the light.

The 5-box structure

Slot	Role	What goes here
A	The common Goal	What both sides agree they're ultimately trying to achieve.
B	The Need behind want D	A legitimate prerequisite for the Goal.
C	The Need behind want D′	A different legitimate prerequisite for the Goal.
D	The Want — one side's chosen action	What B says "to satisfy me, we should…"
D′	The Want — the other side's chosen action	What C says "to satisfy me, we should…"

The conflict is between **D and D′**. Both can't be done simultaneously (that's the mutex). But B and C are both real needs, and A is a shared goal — so the conflict can't be resolved by picking a side. It can only be resolved by finding the assumption that makes D and D′ feel mutually exclusive, and then *not making that assumption anymore*.

Visualized, the EC looks like a flat tree on its side:



Read it in either direction. **Goldratt's preferred read** is A-first (top-down): *"In order to achieve A, we must satisfy B. In order to satisfy B, we must do D. In order to achieve A, we must also satisfy C. In order to satisfy C, we must do D'. And D conflicts with D'."* The whole point of the diagram is the awkwardness of that last sentence.

The method, neutral of tool

1. **Name the conflict in shorthand.** "We need to ship fast vs. we need to ship reliably." "Engineering wants stability vs. Product wants velocity." Whatever the room would say. This isn't the EC; this is the *cover label* for the conversation.
2. **Find D and D′ — the two wants.** Concrete. Action-shaped. "Ship the feature this sprint" / "Defer the feature one sprint." Not "be faster" / "be more careful." The wants are what people actually advocate when they argue.
3. **Find B and C — the two needs.** For each want, ask: *why do you want that?* The answer is the need. The need isn't the next action; it's the *condition the want is in service of*. B = "Maintain market position by shipping ahead of competitor X." C = "Avoid eroding customer trust through buggy releases."
4. **Find A — the shared goal.** Both B and C are conditions for what *bigger* thing? The answer is the goal. Almost always: "long-term commercial success of this product," "customer satisfaction," "team sustainability." If you can't find a shared A, the conflict is at a higher level than you've named it; back out and re-frame.
5. **Name the mutex.** Why do D and D′ conflict? Often "we only have one engineering team's time this sprint." Sometimes "you can't simultaneously have stable and not-yet-tested code in the same release."
6. **List the assumptions on each arrow.** Every necessity arrow ("B requires D") rests on assumptions. Write them down. There will be more than you expect.
7. **Find the breakable assumption.** One of those assumptions, when challenged, is false or no-longer-true. Naming it is the *injection*. The cloud evaporates: it turns out D and D′ aren't actually mutually exclusive, OR they were both wrong actions for B and C, OR the mutex isn't real.

The worked example

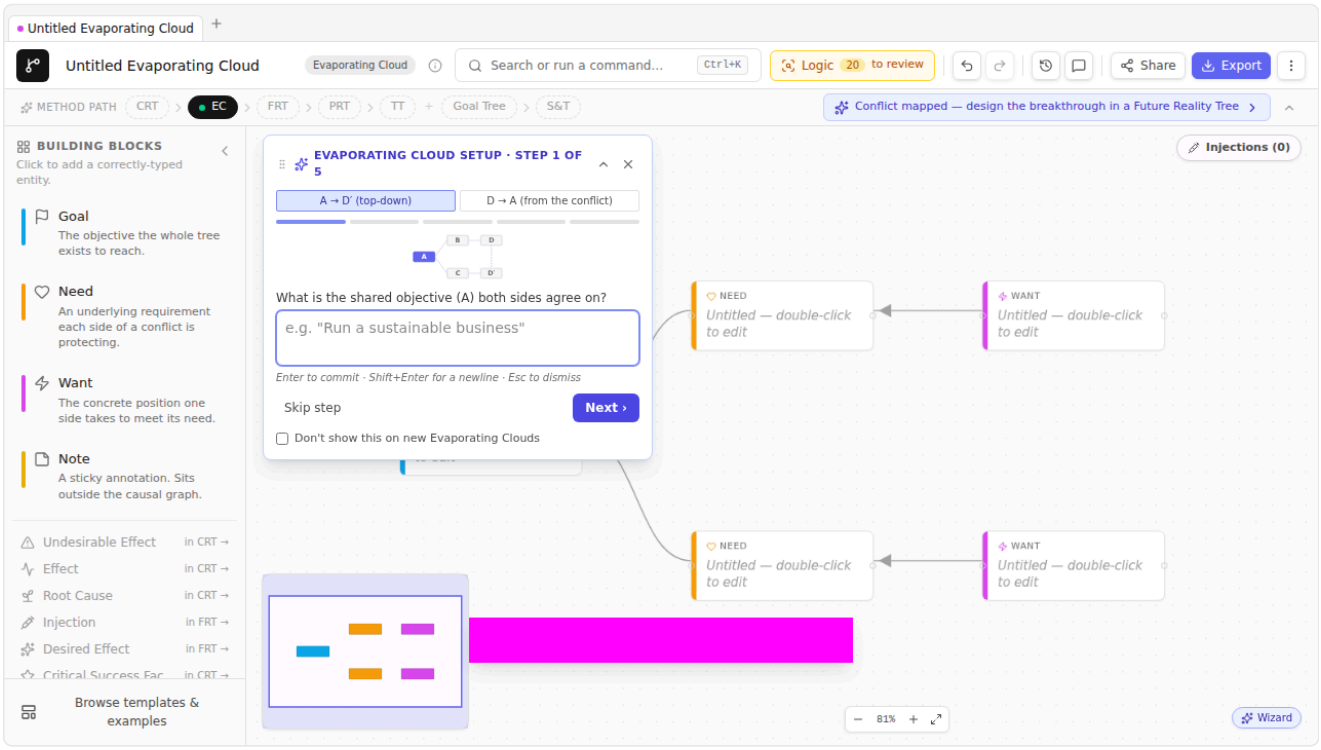
We continue from Chapter 4. The CRT identified "Support lead has no protected drafting time" as the core driver. The natural follow-up question: *why has that persisted?*

Because the support team can't actually spare anybody. Every hour the lead spends drafting is an hour fewer on the queue, the queue grows, customers complain, leadership escalates, lead is back on the queue. The system has been in this loop for months.

Underneath: a conflict. Let's draw it.

Step 1 — Start the EC

`Cmd+K → New diagram → Evaporating Cloud`. The Creation Wizard opens at step 1, prompting for **A — the common goal**. The canvas is pre-seeded with five empty boxes in the canonical 5-box layout.



🛠 **How TP Studio helps:** The EC creation wizard is a 5-step guided dialog that fills the slots in order (A → B → C → D → D′). Each step commits the current entity live, so partial dismissals leave the canvas in a useful state. The Wizard's "from the conflict" toggle reverses the order to D-first (D → D′ → B → C → A) — useful when the cover-label conflict is what surfaced first in conversation and the goal is the thing you haven't yet articulated.

Step 2 — Write A

The common goal. For our example: **A sustainable support function**. Type it, click Next.

The wizard auto-positions A at the canonical left-center position and gives it the sky-blue Goal stripe.

Step 3 — Write B (need behind want D)

The wizard prompts: "What is the first need that the goal requires?" Think about who's been advocating *not* protecting drafting time. That's leadership. Why? Because they need responsiveness to keep customers retained while they're already losing some. So B = **Keep ticket queue responsive to retain at-risk customers**.

Step 4 — Write C (need behind want D′)

"What is the second, conflicting need?" The advocate for protected drafting time would be — also the support lead, also you-the-analyst. Why? Because without the canonical answer base and the triage rubric, the team can't scale; you'll be in the same loop in six months. C = **Build durable structure (rubric + answer base) so the team scales**.

Both needs are legitimate. Both serve A.

Step 5 — Write D

"What is the first action that satisfies need B?" The current behavior is: **Support lead stays on the queue**. That's what's been happening for the last six months.

Step 6 — Write D′

"What is the conflicting action, satisfying need C?" **Support lead takes 2 days/week off the queue to build structure**.

Both wants are concrete actions. They conflict because the lead is one person — there isn't a both-at-once.

Step 7 — Mark the mutex

The wizard closes; the canvas now shows the 5-box EC. Click the edge between D and D′ (if it exists; otherwise, draw it). Inspector → Mutual exclusion toggle → on. The edge turns red with a ⚡ glyph. The `ec-missing-conflict` validator stops firing.

Step 8 — Verbalize

Open the **VerbalisationStrip** above the canvas (it's there by default on EC docs). It renders the cloud as a paragraph:

> *In order to achieve A sustainable support function, we must keep ticket queue responsive to retain at-risk customers. In order to keep ticket queue responsive to retain at-risk customers, support lead stays on the queue. In order to achieve A sustainable support function, we must also build durable structure (rubric + answer base) so the team scales. In order to build durable structure (rubric + answer base) so the team scales, support lead takes 2 days/week off the queue to build structure. And these two wants conflict.*

Read it aloud. It should sound exactly like a problem somebody at the company would describe. If it doesn't, the cloud isn't the right one.

Try the two-party variant. Document Inspector → EC Verbal Style → `twoSided`. Now the strip reads:

> *In order to achieve A sustainable support function, leadership wants to keep ticket queue responsive. To do so, they want support lead stays on the queue. In order to achieve A sustainable support function, the team wants to build durable structure. To do so, they want support lead takes 2 days/week off the queue. And these two wants conflict.*

The two-sided framing makes the political contour of the cloud visible. Useful in workshops where it matters who-said-what.

Step 9 — Surface the assumptions

This is where the cloud evaporates.

For each of the four B→D, C→D′ necessity arrows, list assumptions. TP Studio's **Assumption Well** in the Edge Inspector lets you add them as edge annotations with a status you set directly (`unexamined` / `valid` / `invalid` / `challengeable`); each renders as a violet card linked to its arrow, and pressing `A` with the edge selected adds

one fast. When an assumption is anchored to an edge, TP Studio draws a faint dashed grey line on the canvas from the assumption entity to the midpoint of the edge it pertains to — so even with multiple assumptions in play you can see at a glance which arrow each is challenging without opening the inspector.

Pick the **B → D** arrow ("To keep queue responsive, support lead must stay on the queue"). Click the edge. The Assumption Well shows up in the Edge Inspector. Add:

- *Only the support lead can resolve the hard tickets.*
- *Tickets sufficient to keep the queue responsive must all be resolved by humans.*
- *We can't temporarily reduce inbound ticket volume.*
- *No structural improvement could pay off within the timeframe leadership cares about.*

Pick the **C → D′** arrow. Add:

- *Building the structure requires the lead specifically — no other agent can do it.*
- *The structure must be built in dedicated blocks, not incrementally.*
- *Lead time on the structure is more than the timeline before the next renewal cycle.*

Read each one aloud. Some are clearly false; some are clearly true; one or two are *the* assumption — false but never previously named, and false-ness of which dissolves the conflict.

In our example: **"Only the support lead can resolve the hard tickets"** is the key. If two L2 agents could be trained on the hardest 20% of tickets, the lead's queue time drops, freeing the drafting time, *without* sacrificing queue responsiveness. Both wants get satisfied. The mutex was a habit, not a fact.

That false assumption is the **injection**. In the Inspector's InjectionWorkbench, mark it as `valid: false` (i.e., we've decided this assumption is wrong). Add a new entity titled **Train 2 L2 agents on the hardest 20% of ticket types** and mark it as type `Injection`. Link it to the assumption.

The cloud has evaporated.

Step 10 — Stop

The EC is done when:

- All five slots (A / B / C / D / D′) are concrete and read naturally aloud.
- The mutex is flagged on the D ↔ D′ edge.
- Each B→D, C→D′ arrow has at least two assumptions written down.
- At least one assumption per arrow has been actively classified (valid / invalid / open).
- One assumption has emerged as **the breakable one**, with an injection drafted that resolves the conflict.
- You can describe the resolution in one sentence without rationalizing: *"Once we have two L2 agents who can handle the hard tickets, the lead can take protected drafting time without queue slippage."*

Cloud progression — the same tool, escalating roles

Cohen frames the cloud not as a single artifact but as a *progression*. The same five-box structure does different jobs as the analysis deepens:

- A **UDE cloud** sits behind a single undesirable effect — the dilemma that keeps *this* problem alive.
- A **Consolidated cloud** merges several UDE clouds that share a shape.

- A **Core cloud** is the recurring conflict underneath many UDEs at once — the one at the base of the CRT, and the one worth breaking.

A fourth turns up constantly in practice: the **Firefighting** (Lieutenant) cloud — *patch the symptom now* vs. *stop it coming back* — the trap that keeps an organisation reacting instead of improving.

In TP Studio this is an optional **Cloud type** label (Document panel → *Cloud type*, on EC documents): Dilemma, Conflict, UDE, Consolidated, Core, or Firefighting. It's just a label — nothing about drawing or reading the cloud changes — but tagging one drops a small chip by the title so a folder of clouds reads as a progression rather than a pile. The Templates library (`Cmd/Ctrl+K → Browse templates…`) ships a *UDE cloud*, a *Core cloud*, and a *Firefighting cloud* as pre-tagged starting points.

The resistance cloud — why people both want and fear change

There's one cloud every change effort eventually walks into, and it's worth drawing *before* you need it. Ask a manager how people react to change and you'll hear a contradiction: employees **want** it — they're bored by routine, they chase variety, autonomy, promotion — and they **resist** it, often in the same breath ("happy and scared" about the very same promotion). The organizational psychologist Efrat Goldratt-Ashlag mapped that contradiction as a cloud, and it explains a great deal of what otherwise looks like irrational foot-dragging.

The goal both sides serve is plain: **be happy at work**. What makes the cloud bite is that this goal rests on two *different* needs, and each one pulls the opposite way on change:

- **Satisfaction** — a sense of achievement. And achievement, by definition, means pulling off something you weren't certain you could. No novelty, no challenge, no achievement. So *to get satisfaction, you have to embrace change.*
- **Security** — and here's the sharp redefinition: security is **confidence in the reliability of your own predictions**, not their content. You feel secure when you can foresee what's coming in the areas you care about. Change injects unfamiliar elements, which lowers that reliability. So *to protect security, you resist change.*

Slot	The resistance cloud
A — goal	Be happy at work
B — need → **D** — want	Get satisfaction (a sense of achievement) → **embrace the change**
C — need → **D′** — want	Feel secure (trust my predictions hold) → **resist the change**

The two wants are *embrace change* and *resist change* — a near-perfect mutex, and both are legitimate. That symmetry is why the same person feels both pulls at once, and why "just explain the benefits again" so reliably fails.

Two facets sharpen it:

- **The doubt sweet-spot.** Achievement needs *some* doubt, but not too much. Too little and the task is a foregone conclusion (no pride in clearing it); too much and people give up before they start. Satisfaction lives in the narrow band between "trivial" and "hopeless."
- **Security is content-blind.** You can confidently predict something *bad* and still feel secure in the prediction itself — the dread you feel then is lost *satisfaction*, not lost security. Insecurity only creeps back when the bad prediction starts undermining your *other* predictions ("if the budget's cut, can I still predict I'll keep my team?").

Breaking it — the two channels

The payoff of drawing this cloud is that it tells you exactly where the injections live. You don't argue people out of resisting; you address the *need* their resistance is serving. There are two channels, one per need:

1. **Protect security.** Resistance is a rational defense of prediction reliability — so don't let it crash. Find the areas the person genuinely cares about, then supply the information and training that keep them able to forecast *through* the change. Someone who still feels they can predict what's happening doesn't experience the change as a threat.
2. **Offer satisfaction.** Give people a real, owning role in the change itself — a decision, a piece of the plan, something that's theirs — in an area they value. Now the change *is* the achievement, and the satisfaction need flips from fighting the change to driving it.

In TP Studio terms, each channel is an **injection** that breaks an assumption on the cloud: channel 1 attacks *"this change must reduce my ability to predict"* (the assumption under the D′→C arrow); channel 2 attacks *"this change is something done to me, not by me"* (under D→B). Open **Browse templates…** and drop in *Resistance to change — Efrat's cloud* — it arrives with those two channels already pinned as dotted **notes** beside the needs they protect (notes are non-causal, so they don't disturb the cloud's logic or the CLR checks). List the assumptions on the two arrows, and the injections almost write themselves. From there it's the standard chain: carry each injection into an FRT and check it actually buys cooperation without spawning a negative branch — change-fatigue is the usual one.

> 💡 ***Why this matters in the room:*** *when you hit resistance, the instinct is to push harder or sell the benefits louder. This cloud says both instincts miss — resistance is defending* security *(prediction reliability), while benefits speak only to* satisfaction*. Naming which need is in play tells you which of the two channels to reach for.*

Rapid 3-cloud diagnosis

Sometimes you're nowhere near a finished Current Reality Tree — the team is standing around a whiteboard, three problems are bouncing off each other, and someone says "what is actually going on here?" That's the moment for the **Rapid 3-cloud diagnosis** wizard: a two-step guided overlay that builds a Core Cloud from raw frustrations in a single sitting, without a full CRT.

Invoke it from the command palette: `Cmd+K → "Rapid 3-cloud diagnosis…"`. An overlay opens, separate from whatever you already have on screen; nothing you were working on is touched.

Step 1 — Capture three UDE conflicts

The first step of the overlay presents three side-by-side slots. For each of the three undesirable effects (UDEs) you want to explain, you give it two things: a brief label for the UDE itself, and the conflict you feel underneath it — specifically, **what you DO** (D) versus **what you feel you should do instead** (D′).

Suppose your team's three UDEs are:

1. *Releases slip* — every quarter, the ship date slides by two or three weeks, and leadership scrambles to explain it to stakeholders.
2. *Quality bugs recur* — the same categories of defects appear in successive releases, even after postmortems.
3. *People are burning out* — two strong engineers left in the past year; the rest are running on fumes.

For each, you identify the do-vs-should-do tension:

#	UDE	What you DO (D)	What you feel you should do instead (D′)
1	Releases slip	Push on every deadline regardless of state	Hold capacity back; fix the process before pushing harder
2	Quality bugs recur	Ship as soon as the hot path is green	Reserve a structured integration window before every release
3	People are burning out	Load the team fully to hit commitments	Actively protect slack so the team can invest in themselves and the system

Fill in the six fields. The wizard doesn't ask you to resolve anything yet — it just records the three tensions. Click "Next".

Step 2 — Consolidate to a single Core Cloud

The second step shows the three do-vs-should-do pairs side by side and asks you to find the one cloud underneath all of them. This is the consolidation move at the heart of the method: multiple UDE clouds that share a shape collapse into one Core Cloud.

Look at the D column — *push hard on every deadline*, *skip the integration window*, *load the team fully*. What need does that pattern serve? Something like: **Hit commitments this quarter**. There's real pressure behind it — contracts, stakeholders, sprint goals.

Now look at the D′ column — *protect capacity*, *reserve integration time*, *protect slack for investment*. What need does that serve? Something like: **Keep the team and the system healthy**. Also real — without this, the machine that makes commitments possible eventually seizes.

Both needs are legitimate. Both serve the same shared objective. Type it in the top field: **Deliver reliably and sustainably**. Then name the two needs and confirm the two wants the wizard has already inferred from your D/D′ entries.

When you click "Create Core Cloud", TP Studio:

- opens a new Evaporating Cloud document in its own tab, tagged as **Cloud type: Core**,
- pre-fills the five boxes (A / B / C / D / D′) from the values you entered,
- writes the three UDE labels and their D/D′ pairs into the document description as the source conflicts — preserved for reference, invisible on the canvas itself.

You land on a fully formed, tag-annotated cloud ready to work:

```
A  — Deliver reliably and sustainably
B  — Hit commitments this quarter          →  D  — Push hard on every deadline
C  — Keep the team and the system healthy  →  D' — Hold capacity back to invest
```

The three UDEs — slipping releases, recurring bugs, burnout — all fall out of the same core tension: D and D′ can't both be maximized at once, so the team alternates between them (or freezes, unable to choose), and each UDE is just a different symptom of that oscillation.

From here, it's a standard cloud

The resulting document is an ordinary Evaporating Cloud. Nothing about reading, editing, or breaking it changes:

- Open the **VerbalisationStrip** and read it aloud. Does it sound like the organization talking? If not, adjust the wording in the boxes.
- Mark the **mutex** on the D ↔ D′ edge (Inspector → Mutual exclusion).
- Add **assumptions** to the B→D and C→D′ arrows in the Assumption Well. On D→B ("to hit commitments, we must push on every deadline") you'll quickly find: *Any capacity held back is capacity that doesn't ship*. That assumption is breakable — a team with slack may ship more sustainably than one burning on every sprint.
- Draft the **injection** in the InjectionWorkbench. Mark the assumption invalid and write the resolution: *"If we establish a sustainable pace with explicit recovery cycles, we can hit commitments more reliably than by pushing every deadline — because we stop losing delivery capacity to turnover and technical debt."*

The Rapid 3-cloud diagnosis is not a shortcut that skips the rigor — it's a faster **on-ramp**. You're still doing the same cloud work; you're just starting from raw frustrations rather than a finished CRT. Once the Core Cloud is evaporated, carry the injection forward into a Future Reality Tree to check it actually delivers the good effects you want, and that it doesn't spawn new problems of its own.

> ***🛠 How TP Studio helps:*** *`Cmd+K → "Rapid 3-cloud diagnosis…"` opens the two-step overlay. Step 1 takes three UDE labels and their D/D′ pairs. Step 2 shows the pairs side by side and prompts for A / B / C. On completion, a new EC document tagged as a Core cloud opens in its own tab; the three source conflicts are preserved in the document description. Nothing on your current canvas is affected.*

Sidebars

> ***🛠 How TP Studio helps***
>
> - `Cmd+K → New diagram… → Evaporating Cloud` → *opens the* ***Creation Wizard*** *which guides A / B / C / D / D′ in order.*
> - ***Reading-direction toggle*** *in the Wizard for A-first (default) vs. D-first (when the conflict surfaced first).*
> - ***EC Verbal Style*** *toggle in the Document Inspector:* `neutral` *("we must") or* `twoSided` *("they want / I want") for two-party framing.*
> - ***Cloud type*** *label in the Document Inspector — tag an EC as a UDE / Consolidated / Core / Firefighting cloud (the TP Basics progression); a chip by the title records the role. Three pre-tagged clouds ship in the Templates library.*
> - ***VerbalisationStrip*** *above the canvas — renders the cloud as a paragraph, updates live as you edit.*
> - ***Assumption Well*** *in the Edge Inspector — per-arrow assumption records with* `unexamined` / `valid` / `invalid` / `challengeable` *status and links to injection entities.*
> - ***InjectionWorkbench*** *in the EC inspector — list every proposed injection with* `implemented` *toggles for FRT carry-forward.*
> - ***Mutex (⚡) edge flag*** *on the D ↔ D′ edge. The* `ec-missing-conflict` *validator fires until one such edge exists.*
> - ***EC Workshop Sheet PDF*** *—* `Cmd+K → Export → EC workshop sheet` *— generates a one-page PPT-style layout with the guiding questions baked in. Good for handouts.*
> - `ECReadingInstructions` ***strip*** *above the canvas — the dismissible 1/2/3 numbered hints reminding you of the reading direction. Default-hidden; re-enable via* `Toggle EC reading guide` *.*

💡 ***Practitioner tips***

- ***D and D' must be actions, not states.*** *"We want stability" isn't a want; it's a value. The want is "we want to gate this release on a 2-week soak test." Concrete actions are what people argue about in meetings.*
- ***B and C must be needs, not wants.*** *"Need to be fast" isn't a need; it's a paraphrase of D. The need is what the speed is in service of — usually a market position, a customer commitment, a financial reality.*
- ***A must be genuinely shared.*** *If you can't find an A that both advocates would agree to, the conflict is at a higher level than you've framed it. Back out, ask "and what is that in service of," try again.*
- ***Don't skip the assumption listing.*** *The temptation is to draw the 5 boxes, declare the cloud "done", and walk away. The breakable assumption hides in the assumption list, not in the boxes. The boxes are scaffolding for the listing exercise.*
- ***One EC per conflict.*** *Don't try to fit "Engineering vs. Product vs. Customer Success" into one EC. Draw two — one for the Eng/Product conflict, one for the CS/Eng conflict — and see if they share an A.*

⚠ ***Common mistakes***

- ***Picking a false A.*** *"Make money" is the lazy A. It's usually true but it's also true for every corporate conflict, which means it doesn't help you. The good A is specific enough that it could plausibly not hold ("be the trusted long-term partner for mid-market SaaS customers"). If the A is generic, the B and C will be generic too.*
- ***Equating D with the long-term solution.*** *D is the current behavior (or the chosen-side behavior in a debated decision), not the eventual fix. The eventual fix is the injection you draft after dissolving the cloud. Don't shortcut.*
- ***Writing wishful assumptions.*** *"We could just hire more people" is not an assumption that dissolves the cloud — it's a wish. Assumptions are claims you could plausibly evaluate as true/false right now.*
- ***Skipping verbalisation.*** *If you can't read the cloud aloud and have it sound like a problem your colleagues would recognize, the cloud isn't real. Re-frame.*

🛑 ***When to stop***

- *The cloud reads aloud as a real, recognizable problem.*
- *Each necessity arrow has assumptions, classified.*
- *One breakable assumption is named.*
- *An injection exists addressing the breakable assumption.*
- *The resolution sentence — "Once X, the conflict dissolves because Y" — is one sentence and stands up to a "really?" challenge.*

✏️ ***Now you try.*** *Find a decision your team keeps relitigating — a chronic either/or. Open an EC (`Cmd+K → New diagram… → Evaporating Cloud`) and fill the five boxes: the shared goal (A), the two needs (B / C), the two conflicting wants (D / D'). Mark the D ↔ D' conflict, then add the assumptions on each arrow and cycle their status chips until you find one that's* ***Invalid*** *— that breakable assumption is your injection.*

🔁 **Chain to next:** the EC names the conflict and drafts an injection. The Future Reality Tree checks whether the injection actually delivers the desired effects without spawning new UDEs.

→ Continue to Chapter 6 — Future Reality Tree

Chapter 6 — Future Reality Tree

What would it look like solved?

> 🎯 ***What this process is for*** *A Future Reality Tree (FRT) tests whether your proposed injections actually solve the problem. It answers: "If I make these changes, what will the system produce instead?" The same causal model as a CRT, but starting from injections instead of root causes and reaching upward to desired effects instead of undesirable ones. The point is prediction: if the FRT is logically sound, the injections will work; if it isn't, they won't.*

The premise

A CRT diagnoses. An EC names the conflict and cracks it. An FRT designs the solution and tests it. Each is a deliberate step; skipping straight from EC to "let's implement the injection" loses the discipline of *checking your work before the rollout*.

The check matters for three distinct reasons.

First: injections have second-order effects. The thing you do to fix UDE-1 might cause UDE-7. In complex systems — and any system with more than a handful of people qualifies — the side-effects of an intervention often outweigh the intended effect. Goldratt called the unanticipated downstream UDE the **Negative Branch**, and the FRT is structured to make Negative Branches visible before the rollout, not after. The NB search is not a risk-management add-on; it is the core value proposition of the diagram.

Second: prediction and hope are not the same thing. A project plan expresses *intent* — "we will do X, then Y, then Z." An FRT expresses *causal claims* — "because X, Y will follow; because Y, Z will follow." The difference is that causal claims are falsifiable. You can disagree with a cause-effect arrow in an FRT; you cannot disagree with an action item in a Gantt. Drawing an FRT forces the team to surface the assumptions behind the chain, agree on which ones are solid, and name the ones they're betting on. That naming process is where most of the value lives.

Third: FRTs reveal reinforcing loops. A good injection doesn't just eliminate the current UDEs — it sets off a positive feedback cycle. The triage rubric the lead finally writes doesn't just reduce resolution time; it lets junior agents handle tickets they couldn't handle before, which reduces escalations, which frees senior time, which accelerates the next improvement. Drawing the loop makes the compounding explicit. It also makes the *turning point* explicit — the moment when the investment starts paying back faster than it's being spent, and the team stops feeling like it's swimming upstream.

The FRT is the only Thinking Process diagram that lets you read a plausible future as a logical argument. That's its power, and the discipline it demands: every edge is a bet, and you should know exactly what you're betting.

The method, neutral of tool

1. **Start with the injections you drafted in the EC**, marked as `Injection` type. They are the bottom of the FRT — the things you will *do* or *introduce*. If you have multiple injections from the EC's InjectionWorkbench, bring them all; their independence (or interdependence) will become visible in the tree.
2. **Above each injection, list the immediate desired effects you expect.** "The team has protected drafting time." "The triage rubric exists." "Senior agents stop context-switching." Mark each as `Desired Effect`. Keep

them to one concrete, observable thing per entity — not "everything improves" but "resolution time drops below 4h."

3. **Connect upward — sufficient causality, same grammar as the CRT.** "Because of injection X, desired effect Y." Verbalize each edge as you add it. If the sentence sounds weak — "because X, Y might happen, kind of" — the edge needs an AND companion or the claim needs tightening.
4. **Build the full chain to the elimination of each original UDE.** The FRT is correct when every CRT UDE has a path *down* to one or more of your injections. If a UDE is left dangling — no injection reaches it — either you need another injection or you misidentified the UDE's root cause in the CRT.
5. **Hunt for Negative Branches.** For each major desired effect, ask: "What new, bad thing might this *also* cause?" Don't just think about it — write each one down as a UDE entity and draw the branch. The act of drawing forces precision. "Training takes time" is a vague worry; "Queue load on existing seniors spikes by 30% during L2 training — SLA breaches increase" is a Negative Branch you can actually address.
6. **Trim each Negative Branch.** For every negative-branch UDE, draft a *trimming injection* whose whole job is to break the causal chain before the bad effect follows. The trimming injection is wired to the UDE with a negative-polarity edge: "because this injection is in place, the bad effect does *not* follow." Name it concretely — not "mitigate the training impact" but "hire one temp contractor for 8 weeks to cover queue during training."
7. **Flag Positive Reinforcing Loops.** When a desired effect causes something that causes the original desired effect more strongly, you've found a virtuous cycle. Tag the loop-closing edge as a back-edge in the Edge Inspector. Loops don't invalidate the diagram — they enrich it. They also raise a question you should answer explicitly: *when* does the loop become self-sustaining? That threshold is usually the first milestone worth measuring in the rollout.
8. **Verbalize the FRT** the same way you verbalized the CRT. Step through every edge with `Cmd+K → Start read-through`. If a chain reads as a plausible future — not as a wish-list — the FRT is ready. If a chain reads like hope dressed as prediction, find the assumption that's doing the load-bearing work and mark it explicitly on the edge.

The worked example

We continue from Chapter 5. The EC identified the breakable assumption and produced one primary injection: **Train 2 L2 agents on the hardest 20% of ticket types**.

Step 1 — Open the FRT and bring the injection across

`Cmd+K → New diagram… → Future Reality Tree`. Empty FRT canvas opens.

You could create the injection by double-clicking and typing. But if you're working in the same session where the EC is open in an adjacent tab, there's a faster path: select the injection entity in the EC, then `Cmd+K → Carry this into a new FRT…`. TP Studio creates a new FRT document with the injection already placed and linked back to its EC source.

Mark or confirm its type as `Injection` via the Inspector. The stripe turns emerald.

Step 2 — First-order desired effects

Ask: *if this injection lands, what will be true immediately — before any downstream chain fires?*

Three things follow directly:

- **Lead's queue load drops ~30%.** The L2s now handle a tranche of tickets that previously escalated to the lead.

- **Lead has 2 days/week of protected drafting time.** This is the direct expression of the cloud's dissolution: the conflict between queue responsiveness and drafting time no longer forces a choice.
- **L2 agents have a clearer career-development path.** Bonus desired effect — worth noting even if it wasn't in the original UDE list. The FRT often reveals value you weren't explicitly chasing.

Add each as a `Desired Effect` entity (indigo stripe). Connect each upward from the injection.

Step 3 — Build the causal chain to UDE elimination

Now build upward, asking "and what does that cause?" at each layer.

The first chain:

- Because lead has protected drafting time → **triage rubric exists** → resolution time drops below 4h on average → **SLA met on >90% of tickets** → customer SLA expectations are met → **customers stop churning at renewal**.

The second chain:

- Because lead has protected drafting time → **consolidated answer base exists** → agents stop redoing prior work for common questions → **cost per ticket drops**.

The third chain:

- Because triage rubric exists AND agents have consistent guidance → **customers get consistent answers** → **NPS trend reverses**.

You now have three of the original CRT's UDEs addressed by traceable chains back to the single injection. Read each chain aloud. *"Because the lead has protected drafting time, a triage rubric exists. Because a triage rubric exists, resolution time drops below 4h. Because resolution time drops below 4h, SLA is met. Because SLA is met, customers stop churning."* Sounds right.

Notice that two chains share the "lead has protected drafting time" node. That convergence is structural good news: a single intervention is doing double duty. But it's also a fragility to mark: if the drafting time gets eroded — by a hiring spike, a product launch, any unexpected inbound surge — both chains fail simultaneously. The FRT makes that dependency legible.

Step 4 — Hunt for Negative Branches

Now slow down. Walk through each major desired effect and ask: "What else might this cause? What could go wrong?"

From "Lead has protected drafting time":

- While the lead is off the queue, queue load on existing senior agents increases. If the L2 training hasn't fully landed yet, seniors handle both their normal escalations *and* the gap the lead left. UDE: **"Queue load on existing seniors spikes during the transition period."**

From "L2 agents now handle hard tickets":

- If the training program was rushed or shallow, L2 agents will misclassify some hard tickets and give customers incorrect answers. The pattern "every agent gives a different answer" — which the triage rubric was meant to fix — resurfaces, but now at L2. UDE: **"L2-resolved tickets are lower quality than L3-resolved ones — customer trust erodes."**

From "consolidated answer base exists":

- If the answer base is built by the lead in isolation and not validated with the broader team, agents will distrust it, stop using it, and fall back to tribal knowledge. UDE: **"Answer base is not adopted — agents revert to ad-hoc answers."**

Add all three as `Undesirable Effect` entities. Draw the branch from the relevant desired effect down to each UDE. The canvas now looks less clean — it should. A clean FRT is usually an incomplete one.

Step 5 — Trim the Negative Branches

For each NB, draft a trimming injection.

NB1 (queue spike): Select the UDE. `Cmd+K → Trim this branch (add a trimming injection)`. TP Studio mints a trimming injection wired to the UDE with a negative edge. Name it: **"Engage one temp contractor for 8 weeks to cover L3 queue during L2 ramp."** This injection negatively causes the queue spike — it doesn't prevent the NB from starting, but it breaks the causal chain so the bad effect doesn't follow.

NB2 (quality drop): Trim. Name: **"Run a 2-week L2/L3 shadowing program before L2s handle hard tickets solo."** This injects a quality gate between the training injection and the solo-handling capability.

NB3 (answer-base adoption): Trim. Name: **"Co-author the answer base in 4 team workshops — agents contribute and own the content."** Ownership, not mandate, is the mechanism. When agents helped build it, they use it.

Three trimming injections, each named to say *what* breaks the branch, not just "mitigate the risk."

Step 6 — Check for Positive Reinforcing Loops

Look for the virtuous cycle that your injections might set off.

Here's one: the triage rubric and consolidated answer base together reduce escalation volume. Reduced escalations mean the lead's queue load stays low even as the business grows. A lower ongoing queue load means the lead can *keep* taking drafting time — not just for the initial build, but for continuous improvement. That continuous improvement produces more structure, which reduces escalations further.

The loop closes: "Lead has protected drafting time" causes "consolidated answer base improves over time" causes "escalation volume stays manageable" causes "lead keeps having protected drafting time."

Add an edge from the "escalation volume stays manageable" effect back up to "Lead has protected drafting time." Select this loop-closing edge and turn on the **Back-edge toggle** in the Edge Inspector. The edge renders differently — a curved back-arc — marking it as the loop-closing move. Tag the group containing the loop with the **Positive Reinforcing Loop** group preset (emerald).

Step 7 — Verbalize and stop

`Cmd+K → Start read-through`. Walk every edge.

The FRT is done when:

- Every CRT UDE traces down to at least one injection.
- Each injection has at least one desired effect above it.
- You've genuinely looked for Negative Branches at each major desired effect — not just gestured at the question.

- Each NB has either a trimming injection that suppresses it, or an explicit "we accept this risk" note on the entity.
- Any reinforcing loops are tagged with back-edges.
- The read-through sounds like a plausible future, not a marketing document.

You now have 4 injections (one primary, three trimming), 5+ desired effects, 3 negative-branch UDEs each trimmed, one reinforcing loop, and a clean path from the injections to the elimination of the original 3 CRT UDEs.

A second worked example — a simpler case

Not every FRT is this layered. Consider a lighter problem: a product team whose sprint velocity keeps undershooting. The EC uncovered a single injection: **Cap WIP at 3 items per developer per sprint, enforced on the board**.

The FRT for this is sparser. One injection directly causes:

- **Stage queues drain to 1-2 items per sprint.**
- **Engineers finish tickets before pulling new ones** — the "start to finish" shift that WIP-cap produces.

Both flow upward to:

- **Hand-off friction becomes visible at the cap line** — because the queue is thin enough that blockers surface while there's still time to act.

Which causes:

- **Per-ticket lead-time variance narrows.**

Which causes:

- **Lead time drops below two weeks at p95.** (Desired effect, measurable, the one the team actually cares about.)

The NB hunt: with a WIP cap, what could go wrong? If the cap is set too aggressively, developers idle while waiting for review. NB: **"Developers block on review wait — perceived productivity drops."** Trim it: **"Add a daily async review slot (15 min) as a forcing function."**

That's a 6-entity FRT with one trimming injection. It's not shallow — it's *appropriately sized for the problem*. The depth bar for an FRT isn't the number of nodes; it's whether you've traced the causal chain honestly and looked for what could go wrong.

Trimming a negative branch

Finding a negative branch is only half the move. Once you've drawn the unintended UDE that your injection would also cause, the corrective answer in Goldratt's grammar is to *trim* it — to add a second injection whose whole job is to break the link so the bad effect won't follow.

TP Studio makes that a single gesture. Select the undesirable effect at the tip of the branch and run **"Trim this branch (add a trimming injection)"** from the palette. It mints a trimming injection wired to that effect with a negative-polarity edge — the formal "inject this, and the bad effect doesn't follow" construction, rendered so the polarity is unmistakable. It's one undoable step, so trimming costs you nothing to try.

All that's left is to name the new injection to say *what* breaks the branch — the contractor backfill, the shadowing programme, the co-authoring workshops — turning a spotted risk into a concrete countermeasure on the canvas.

A trimmed FRT is a *realistic* FRT. An untrimmed one is a plan drawn by someone who's never had a plan fail.

> 🛠 ***When the branch deserves its own canvas.*** *The in-FRT Negative Branch group is right for a risk you'll trim in place. When one risk warrants a full analysis — several contributing effects, its own review conversation, a risk register to export — spin it up as a standalone* ***Negative Branch Reservation*** *document (* `Cmd/Ctrl+K → New diagram… → Negative Branch Reservation` *). An NBR's palette is injection / effect / UDE / desired effect, its default type is UDE (the branch's point), and two NBR-specific validators keep the shape honest:* `nbr-no-negative-branch` *reminds you a branch with no UDE still reads as an FRT, and* `nbr-ude-disconnected` *flags a UDE that doesn't trace back to the injection under scrutiny. The* ***Risk Register CSV*** *export pairs with it — each UDE becomes a risk row, its trimming injection inferred as the mitigation. Five NBR starter templates ship in the Templates library.*

Prediction vs. hope — the central discipline

The distinction between prediction and hope deserves a full paragraph because it's where most FRTs go wrong.

A *hope* statement looks like a cause-effect claim but isn't: "Because we train L2 agents, customer satisfaction will improve." There's a causal structure implied, but the mechanism is missing. *How* does training lead to satisfaction? Via what intermediate steps? Under what conditions?

A *prediction* statement makes those intermediate steps explicit and therefore checkable: "Because L2 agents are trained on the hard 20% of ticket types, they handle those tickets without escalation. Because escalations to the lead drop, lead time on hard tickets falls below the SLA threshold. Because SLA is met, the specific customers who were churning due to SLA misses stay at renewal."

The difference isn't semantic. The prediction exposes three assumptions you can evaluate right now: (1) training will actually produce competency on the hard tickets, (2) the SLA threshold is the actual churn driver (and not, say, price), (3) the customers at risk are the SLA-sensitive ones specifically. Each is falsifiable. If any one is wrong, the chain breaks at that link, and you can fix the FRT — and the rollout plan — before you've deployed anything.

The practical test: can someone disagree with a specific edge in your FRT? If every edge feels self-evidently true and no one pushes back, you're drawing hope. Good FRT edges make people say "wait, does that actually follow?" — and that conversation is the value.

FRT scope and the "desired future state" question

One thing practitioners find uncomfortable about the FRT: *how far forward do you draw?* The CRT had a natural stopping point (root cause reached). The EC had a natural stopping point (injection drafted). The FRT, in principle, could go on forever — every desired effect causes more desired effects.

The practical answer is this: **draw to the elimination of your CRT's UDEs, plus one more layer.** That extra layer serves two purposes. First, it confirms the desired effects are actually desirable in their downstream consequences — sometimes an intermediate effect that sounds good causes a problem one step further up. Second, it usually reveals the reinforcing loop, which is worth drawing.

Beyond that, stop. The FRT is a *design tool*, not a utopian scenario exercise. When the tool starts generating "and then everything will be wonderful forever," it has stopped being causally disciplined and started being motivational. Motivational is fine for a vision deck; it's not useful for checking your work.

The Document Inspector's "System Scope" section lets you write a short statement of what the desired future state *is*. Do that before drawing. A one-sentence scope — "Support team meets SLA on 90%+ of tickets, at current head-count, within 6 months" — gives you a terminus. When the FRT reaches the entities that express that scope, it's done.

Sidebars

> ***🛠 How TP Studio helps***
>
> - `Cmd+K → New diagram… → Future Reality Tree` *to start fresh;* `Load example…` *for a reference FRT.*
> - `Cmd+K → Carry this into a new FRT…` *— from a selected injection in an EC, spawns a new FRT with the injection pre-placed and cross-linked.*
> - ***Group presets****: Negative Branch (rose), Positive Reinforcing Loop (emerald), Archive (slate, collapsed) — Group Inspector → Preset. Catalog in* `src/domain/groupPresets.ts` *.*
> - ***Back-edge toggle*** *in the Edge Inspector — marks the loop-closing edge of a reinforcing cycle. The edge renders as a back-arc so the loop is visually unmistakable.*
> - ***Edge polarity*** *—* `Cmd+K → Cycle edge polarity` *or the Edge Inspector's polarity picker. A negative-polarity edge means "this injection* prevents *the effect" — essential for trimming injections. A polarity badge on each edge makes suppression relationships visible at a glance.*
> - `Trim this branch (add a trimming injection)` *— select the UDE at a negative branch's tip; mints a trimming injection wired to it with a negative edge, in one undoable step.*
> - `Start Negative Branch from selected entity` *— palette command (also right-click context menu); creates a rose-coloured Negative Branch group rooted at the selected entity. Use it to keep NB sub-trees visually separated within the FRT.*
> - ***Injection Flower*** *— select an injection entity, then* `Cmd+K → View the injection flower (desired effects · negative branch · plan)` *. The flower dialog shows all three cross-doc petals (FRT / NBR / PRT) linked to this injection, surfacing gaps: an injection whose "negative branch" petal is empty hasn't been vetted for risks.*
> - `Speculate: what changes if… (what-if overlay)` *—* `Cmd+K → Speculate: what changes if…` *lets you set a hypothetical state on any entity and see the downstream cascade propagated across the FRT without committing the change. Useful for "what if the L2 training doesn't land — which desired effects collapse?"*
> - `Start read-through` */* `Read entire diagram at once` *— two verbalisation modes for the FRT. Read-through steps edge by edge (the discipline pass); read-all-at-once generates the full causal narrative in one view (useful for sharing with a stakeholder who wants the logic in one scroll).*
> - ***Annotation numbers*** *(Settings → Display → "Show annotation numbers") — assigns a* `#N` *badge to each entity. With numbers on, you can reference "injection I1" in a rollout plan and link back to the entity it describes. The annotation badge is also how cross-document entity references (* `#42` *) resolve.*
> - ***CLR walkthrough****:* `Cmd+K → Start CLR walkthrough` *— fires the relevant validators (sufficiency, clarity, predicted-effect existence). The* `predicted-effect existence` *check is the one that fires most often in FRTs: "does this predicted intermediate effect actually exist in the world you're describing?"*

> 💡 ***Practitioner tips***
>
> - ***Look for Negative Branches actively, not passively.*** *The FRT's value is mostly in catching them. Don't ask once; ask once per major desired effect. Walk through the list: queue behaviour, team morale, adjacent teams' workloads, supplier relationships, customer expectations. Each is a dimension where an unintended consequence might live.*
> - ***The FRT is a draft.*** *It will be wrong about some second-order effects. The rollout will reveal which ones. Keep the FRT open after the rollout and annotate it: "this edge didn't fire because…", "this NB fired even though we had a trimming injection because…". An FRT annotated against reality is one of the most valuable post-mortems you can do.*
> - ***Name injections with verbs.*** *"Train 2 L2 agents" is an injection. "Better training" is not. The verb-first naming convention makes it clear what someone actually has to do, and it makes the FRT easier to read aloud.*
> - ***The bonus desired effect is real.*** *When the FRT reveals a benefit you weren't explicitly chasing — improved agent career development, reduced hiring overhead, earlier customer relationships — write it down. It belongs in the business case for the injection, and it often unlocks stakeholder buy-in that the primary benefit alone wouldn't.*
> - ***Use the Archive group preset for paths not taken.*** *When you draw a Negative Branch and then trim it, the unsuppressed version (injection without trimming) is a considered alternative. Archive it rather than deleting it: select the group, then choose* ***Preset → Archive*** *in the Group Inspector (slate, collapsed). The reasoning stays visible without cluttering the live diagram.*

⚠ ***Common mistakes***

- ***Trivial FRT.*** *If your FRT is a single injection → single desired effect → "and the UDE goes away", you haven't done the work. Real systems have second-order effects; if you didn't find any, you didn't look hard enough. The minimum useful FRT has at least two layers of intermediate effects and at least one NB search attempt (even if no NBs emerge).*
- ***Confusing prediction with hope.*** *"We hope this will lead to improved customer satisfaction" is not a cause-effect claim. Every edge in the FRT is a prediction you should be able to defend: "I believe A causes B because of mechanism M, under conditions C." If you can't state M and C, the edge is hope. Mark the assumption explicitly on the edge using* `Cmd+K → Add assumption to selected edge` *; come back to defend it or restructure the chain.*
- ***Skipping the Negative Branch hunt.*** *This is the single highest-value step in FRT drawing. Skipping it produces FRTs that read like project plans and predict like horoscopes. The NBR hunt should take as long as the initial tree construction; if it felt fast, you weren't trying hard enough.*
- ***Drawing the FRT before the EC is complete.*** *The FRT tests injections; the EC surfaces the conflict and produces the injection. If the injection isn't yet grounded in the conflict structure — if it's a solution you already had before the EC — the FRT will confirm whatever you believed going in. Let the EC's assumption analysis do its job first.*
- ***Not connecting the FRT back to every CRT UDE.*** *The FRT is done when each original UDE has a path down to an injection. If a UDE is unreachable, either there's a missing injection or the CRT was wrong about that UDE's root cause. Either way, the gap is information.*
- ***Letting the reinforcing loop be implicit.*** *"Everyone knows it'll become a virtuous cycle" is not a reinforcing loop; it's a hope. Draw the loop, tag the back-edge, name the mechanism. If you can't draw it precisely, you don't yet understand why it would be self-sustaining.*

🛑 ***When to stop***

- *Every CRT UDE has a path down to at least one injection.*
- *Each injection has at least one desired effect spelled out above it, connected by a sentence you're prepared to defend.*
- *You've looked for Negative Branches against every major desired effect — actively, not by gesture.*
- *Each NB has either a trimming injection that suppresses it, or an explicit "we accept this risk" annotation on the entity.*
- *Any reinforcing loops are tagged with back-edges.*
- *Verbalisation reads as a plausible future, not a wish-list. You'd be comfortable presenting each edge as a prediction to a skeptical colleague.*
- *The desired future state you wrote in the Document Inspector's System Scope section is visible somewhere in the top layer of desired effects.*

✏️ ***Now you try.*** *Take an injection you believe in — from the cloud above, or any "we should just…". Open an FRT (* `Cmd+K → New diagram… → Future Reality Tree` *), seed the injection, and grow the desired effects forward until you reach the UDEs it should eliminate. Then hunt one* ***negative branch*** *(right-click → Start Negative Branch): what could this fix break? Trim it (Trim this branch). An FRT with no negative branches usually means you haven't looked hard enough.*

🔁 **Chain to next:** the FRT tells you *what* will happen if the injections land. The Prerequisite Tree tells you *what is in the way of making them land* — the obstacles between here and there.

→ Continue to Chapter 7 — Prerequisite Tree

Chapter 7 — Prerequisite Tree

What's in our way?

> 🎯 ***What this process is for*** *A Prerequisite Tree (PRT) surfaces the obstacles between* ***where you are*** *and* ***where the FRT says you want to be****, then pairs each obstacle with an* ***Intermediate Objective*** *(IO) — a state that, if achieved, dissolves the obstacle. It answers: "What has to be true* ***first*** *for the injections to land?"*

The premise

You've diagnosed (CRT), named the conflict (EC), designed the solution (FRT). Now you have to *execute*, and execution runs into obstacles: things you don't have, capabilities you can't yet exercise, dependencies that aren't ready. The PRT makes those obstacles explicit and pairs each with the IO that resolves it.

Goldratt's framing: every injection has a precondition tree underneath it. The PRT *is* that precondition tree, sequenced and obstacle-aware. What distinguishes the PRT from a plain task list is the obstacle. Without naming what's in the way, you're just writing a wishlist. The obstacle is the *reason* the IO is necessary — it converts a wish into a justified step.

This matters more than it sounds. When a real rollout stalls, it almost never stalls because somebody forgot to schedule the work. It stalls because an obstacle that was never named is quietly blocking progress — no budget approved, no second person trained, no agreement from a VP who hasn't been asked yet. The PRT's job is to drag every one of those silent blockers into the open *before* the effort starts, not after.

There is also a structural difference between the PRT and the trees that precede it. The CRT and FRT use sufficiency logic — "A causes B" means A is (at least part of) what's sufficient to produce B. The PRT uses necessity logic — "In order to achieve X, Y must hold." Both read upward toward a goal, but the interpretive frame is opposite. The CRT asks "why does this happen?" The PRT asks "what must be true first?" Keeping that distinction sharp prevents you from accidentally running a sufficiency analysis inside a necessity diagram. If you find yourself writing "because we did X, we now have Y" rather than "in order to have Y, we need X," you've drifted into FRT territory and the PRT isn't the right vehicle.

One more nuance on layout. The PRT shares the same vertical shape as a CRT: the apex sits at the top and the leaves settle at the bottom. Here the apex is the injection (or ambitious goal), and the leaf IOs are the earliest prerequisites — the ones you achieve first. TP Studio's Dagre layout strategy places the injection at the top and the independent leaf IOs at the bottom automatically. But the *sequence* builds bottom-up: you start at the leaves and work upward, each IO unlocking the one above it until the injection becomes reachable. Don't mistake the apex-at-top shape for top-down execution — the bottom of a PRT is where the work begins.

The method, neutral of tool

1. **Start with the injection.** Place it at the top of the canvas — apex at top, leaf prerequisites at the bottom, the same vertical shape as a CRT.
2. **Below the injection, brainstorm obstacles.** For each, ask: "What is currently stopping us from achieving the injection directly?" The obstacle should describe a real *absence* or *barrier*, not a task. "We don't have a triage rubric document" is an obstacle. "Write a triage rubric" is the IO. Keep them distinct.

3. **For each obstacle, pair an IO.** "Two L2 agents reassigned with 40% capacity for training over 8 weeks." The IO is the *state* you need to reach, not the action you take to get there. "Draft circulated and signed off" is an IO. "Schedule the review meeting" is a sub-IO. If the IO is still too large to act on, descend: what obstacle blocks *the IO itself*?
4. **Necessity edges, not sufficiency.** The PRT edge reads: "In order to achieve [upper entity], [lower entity] must hold." TP Studio's default edge type for PRT diagrams is `necessity` — the verbaliser will auto-render it as "In order to obtain…" wording when the read-through overlay is active.
5. **Sequence the IOs.** Some IOs depend on others. Draw dependency edges between IOs: an arrow from IO-A to IO-B means "achieve A before B." A chain emerges from the bottom (independent IOs) upward to the injection. This sequencing is what turns the PRT from a flat list into an implementation plan with a critical path.
6. **Triage span-of-control.** For each IO, ask: is this in your `control` (you can do it unilaterally), `influence` (you need somebody else's cooperation), or `external` (it depends on a party you can't directly influence)? Mark each entity accordingly. The `external` ones need escalation plans — they won't self-resolve.
7. **Stop when the leaves are actionable.** If a leaf IO still has its own obstacle that requires a sub-IO, keep descending. The tree is done when every leaf is something you can genuinely start next week.

The worked example

We continue from Chapter 6. The FRT verified that the primary injection — **Train 2 L2 agents on the hardest 20% of ticket types** — is sufficient to produce the desired effects without spawning unacceptable negative branches. Now we need to get there.

`Cmd+K → New diagram…` → Prerequisite Tree. Empty PRT canvas opens. Add the injection at the top as an `Injection` entity.

Step 1 — Brainstorm obstacles

Below the injection, brainstorm everything that is currently stopping you from performing that training tomorrow:

- *Obstacle:* We don't have a list of "the hardest 20% of ticket types" — nobody's audited the queue.
- *Obstacle:* Both candidate L2 agents have full queues already; there is no slack to take training.
- *Obstacle:* No training curriculum exists.
- *Obstacle:* No budget has been allocated for the contractor backfill needed to create queue slack (this was FRT injection I2).

Add each as an `obstacle` entity via the Inspector's Type grid. TP Studio gives them a rose stripe — the visual contrast from the injection's syringe icon tells you at a glance "this is the barrier layer, not the goal layer."

Step 2 — Pair each obstacle with an IO

For each obstacle, write the IO that dissolves it. Keep the IO as a *state to be achieved*, not an action:

- *IO:* Ticket-type audit complete; "hardest 20%" defined by escalation count + resolution time. (`intermediateObjective` type, blue stripe.)
- *IO:* L2 capacity freed by re-routing approximately 40% of their existing tickets to L1 with escalation rules in place.
- *IO:* Curriculum drafted and reviewed: four modules of two hours each, approved by the support lead.
- *IO:* Budget request approved by Finance for eight weeks of contractor backfill.

Connect each IO to its obstacle (IO → obstacle), and each obstacle to the injection (obstacle → injection). The canvas now shows a wide, flat structure: four IO–obstacle pairs feeding the injection.

🛠 **How TP Studio helps:** Select any `obstacle` entity and run `Cmd+K → Add Intermediate Objective for this Obstacle (PRT)`. TP Studio mints a new `intermediateObjective` entity, opens its title for editing, and connects it to the selected obstacle automatically. You can also use `Cmd+K → Mark entity as Obstacle (PRT)` and `Cmd+K → Mark entity as Intermediate Objective (PRT)` to retype entities you've already written as plain effects.

Step 3 — Add dependency edges

Some IOs depend on others and must be sequenced:

- "L2 capacity freed by re-routing" depends on "Ticket-type audit complete" — you need to know what the hardest 20% looks like before you can decide which tickets to re-route to L1.
- "Curriculum drafted and reviewed" also depends on the ticket-type audit — the curriculum must be calibrated to the actual hard cases, not a guess about them.

Add those edges: IO → IO, the same necessity direction. The PRT now reads: audit first, then in parallel the capacity-freeing and the curriculum, and only once both of those hold can the training actually run.

The "Budget approved" IO has no dependency on the others — Finance doesn't need the audit before they can approve the headcount. So that leaf sits at the same level as the audit, and both can start immediately.

The tree has just surfaced the critical path: audit → (capacity + curriculum in parallel) → injection. Anyone who wants to know "what do we do first?" can read the answer directly off the dependency edges.

Step 4 — Set span-of-control

Click each IO and set span-of-control in the Inspector:

- Ticket-type audit: `control` — the support lead can run this without asking anyone.
- L2 re-routing: `influence` — requires agreement from the two L2 agents and probably their manager.
- Curriculum drafting: `control`.
- Budget approval: `influence` — requires Finance; this is the IO most likely to become a blocker if nobody starts the conversation early.

The `influence` IOs are the ones to watch. Each one names somebody outside your immediate team who needs to say yes. In a real rollout, the budget IO is the one that kills momentum at week three because nobody filed the request at week one.

> 🛑 ***When to stop***
>
> - *Every leaf IO is something you can start next week (or whatever short timescale matters for your context).*
> - *Every IO is paired with a named obstacle it resolves.*
> - *Sequencing dependencies are explicit edges, not implicit.*
> - *Span-of-control is set on each IO so the rollout owner knows what's `control` vs. `influence`.*
> - *You can read the tree top-down aloud, edge by edge, and each "In order to obtain…, we must…" sentence sounds like a real constraint, not a bureaucratic formality.*

Exporting an ordered plan

A PRT carries no explicit step numbers. Its order lives entirely in the dependency edges — the IO another IO depends on *is* the earlier step, even though nothing on the canvas says "1, 2, 3." That's correct for thinking, but it's awkward to hand to someone who just wants the to-do list.

`Export… → Prerequisite plan (CSV)` does the translation: it topologically sorts the tree so that every IO appears prerequisite-first (an IO that others depend on comes before them) and writes one row per Intermediate Objective — `step / objective / overcomes / depends_on / owner / due_date / status / notes`.

That row shape is deliberately spreadsheet- and tracker-shaped: paste it straight into Jira, Trello, or a sheet and you have a backlog ordered the way the analysis says it must be done, with each item still tied to the obstacle that justifies it. The `overcomes` column is what makes it different from any other task export — it answers the question "why is this step here?" without requiring the reader to open the original diagram.

The export appears only on a document that has at least one `intermediateObjective` entity. Ownership and due-date columns are populated from the entity's `owner` field and `dueDate` attribute respectively; the `status` column shows `done` when you've toggled `implemented` on the entity (matching the TT task export convention). This means a PRT can pull double duty as a living tracker — mark IOs done in TP Studio and re-export to refresh the sheet.

A second worked example — product launch gate

The SaaS-support scenario above is a single-injection PRT with one clear critical path. Not every PRT is that tidy. Here is a different shape: a pre-launch PRT with a cross-functional critical path where several `external` IOs require escalation.

Scenario: your team has designed a pricing experiment (FRT injection): ship a segment-specific pricing tier to mid-market accounts. The FRT validated the injection. Now the PRT asks: what's in the way?

Obstacles that surface in a brainstorm with engineering, legal, and marketing:

- Legal hasn't approved the new pricing language for the product surface.
- The billing service doesn't support per-segment price codes.
- The A/B framework can't target by account segment without a feature flag rework.
- Marketing hasn't built the in-app announcement copy or the email sequence.
- Finance requires a formal pricing-authority memo before any tier change ships.

Pair each with an IO:

- *IO:* Legal review complete; pricing language approved and embedded in the UI copy.
- *IO:* Billing service supports per-segment price codes (new column in the price table + API).
- *IO:* Feature-flag service extended to support account-segment targeting.
- *IO:* In-app and email copy complete, reviewed by brand, staged in the CMS.
- *IO:* Pricing-authority memo signed by CFO and VP Product.

Dependency sequencing: the A/B framework change and the billing change can start in parallel, independent of each other. Legal review and marketing copy can also start immediately. But the pricing-authority memo is a dependency of *all of them* — Finance won't sign until both legal and billing confirm capability. So the memo is not a leaf; it depends on legal and billing finishing first.

When you draw those edges, the critical path emerges: billing + legal → memo → (A/B + marketing + pricing UI) → launch. The longest chain runs through the legal review (two weeks) and then through Finance (one week sign-off window). That is where a project manager should focus first — not on the marketing copy, which is internal and fast.

Mark the CFO sign-off IO as `external`. Mark the legal review as `external` (it depends on outside counsel). Mark the A/B framework change as `control`. Now the `influence` and `external` IOs are flagged on the canvas at a glance — exactly the ones that need conversations started this week.

> 💡 ***Practitioner tip:*** *the `external` IOs are almost always the ones that blow up the schedule, because people underestimate how long it takes to get a decision out of someone outside the team. Surface them early, escalate early, and in the PRT make them explicit dependencies of the IOs that need them. Then the project manager can see the critical path without asking.*

Sidebars

> ***🛠 How TP Studio helps***
>
> - `Cmd+K → New diagram…` → *Prerequisite Tree to start fresh.*
> - `Cmd+K → Load example…` → *Prerequisite Tree for a reference doc (product-launch PRT) to study before drawing your own.*
> - ***`obstacle`*** *entity type (rose stripe, Mountain icon) and* ***`intermediateObjective`*** *type (blue stripe, Milestone icon) — PRT-specific entities in the Inspector Type grid.*
> - ***`Cmd+K → Mark entity as Obstacle (PRT)`*** *and* ***`Cmd+K → Mark entity as Intermediate Objective (PRT)`*** *— type-flip the selected entity without opening the Inspector.*
> - ***`Cmd+K → Add Intermediate Objective for this Obstacle (PRT)`*** *— from a selected* `obstacle`*, mints a paired* `intermediateObjective` *entity, opens it for editing, and wires the IO → obstacle edge automatically.*
> - ***Necessity edges by default*** *for PRT diagrams — the read-through verbaliser renders "In order to obtain…, …must hold." wording automatically (PRT and EC share the* `in order to` *causality mode).*
> - ***Dagre layout (apex at top)*** *— PRT auto-lays out with the injection at the apex and IOs descending to the leaf prerequisites. Run layout again after adding new entities to keep the canvas readable.*
> - ***Span-of-control*** *flags (* `control` */* `influence` */* `external` *) in the Inspector — mark each IO and obstacle so the rollout owner can triage at a glance.*
> - ***`Start read-through`*** *(* `Cmd+K → Start read-through` *) — steps through every edge in topological order, rendering each as "In order to obtain [IO], [obstacle] must be overcome." Discipline for verifying the tree before handing it off.*
> - ***`Export… → Prerequisite plan (CSV)`*** *— topologically sorts the IOs prerequisite-first into a* `step / objective / overcomes / depends_on / owner / due_date / status / notes` *row, ready to paste into Jira, Trello, or a spreadsheet. Appears only when the document has* `intermediateObjective` *entities; the* `status` *column reflects the entity's* `implemented` *toggle.*
> - ***CLR pairing checks*** *— two PRT-specific validators enforce the obstacle↔IO discipline:* `prt-obstacle-no-io` *flags an obstacle with no Intermediate Objective overcoming it, and* `prt-io-no-obstacle` *flags an IO that doesn't overcome any obstacle. If a warning fires, the pairing rule below has been broken.*
> - ***Templates*** *(* `Cmd+K → Browse templates…` *) — five curated PRT starters: Prerequisite Tree starter, Database migration, New-market entry, Performance-review rollout, and Zero-defect manufacturing. Load one as a reference shape before drawing your own.*

> 💡 ***Practitioner tips***
>
> - ***Pair every obstacle with an IO.*** *An obstacle without an IO is a complaint. The IO turns "we can't" into "we will, once X." When you're tempted to leave an obstacle un-paired because it "goes without saying," write the IO anyway — the act of writing it often surfaces a dependency you hadn't noticed.*
> - ***Surface the boring IOs.*** *"Get budget approved." "Schedule training time on calendars." "Get VP to sign the authority memo." These are the things that derail real rollouts. They belong in the PRT. The tendency is to focus the tree on the meaty technical work and skip the organizational prerequisites. Don't. The boring IOs are often on the critical path.*
> - ***The IO is a state, not an action.*** *"Send the curriculum draft to the lead" is an action. "Lead has reviewed and approved the curriculum draft" is an IO. The state framing matters because it defines a clear done-criterion — you can't argue about whether the action "counted"; either the state is true or it isn't.*
> - ***Descend until leaves are startable.*** *If an IO still feels too large — "we need executive buy-in" — sub-decompose it into what that specifically means: what does the executive need to know, agree to, or approve, and what must be true first before you can ask? Each sub-element becomes its own obstacle–IO pair. A well-decomposed PRT has leaves that a single person can start on Monday without a meeting.*
> - ***Use Notes for context.*** *When an IO has nuance ("this requires Finance approval and we have a window before the next QBR — missing that window costs six weeks"), capture it in the IO's description or as a* `note` *entity nearby. The note is non-causal and won't affect the CLR checks, but it preserves the timing context that would otherwise live only in someone's head.*
> - ***Read the critical path before handing off.*** *After layout, scan the longest chain from leaf to injection. That chain is what constrains the timeline, no matter how many parallel tracks exist. Communicating the critical path explicitly prevents the team from optimizing the fast parallel tracks while the slow critical one drifts.*

> ⚠ ***Common mistakes***
>
> - ***Skipping the obstacle and going straight to IOs.*** *Without the obstacle, the IO floats — it's not clear why it's necessary. The pairing is the point: the obstacle is the reason for the IO, and the reason is what prevents the IO from being dismissed as gold-plating.*
> - ***Writing IOs as actions rather than states.*** *"Train two L2 agents" is an action. "Two L2 agents certified on the hardest 20% of ticket types" is a state. States give you a binary done-criterion. Actions leave room for the "we started it, does that count?" conversation.*
> - ***Bottoming out at "we need executive buy-in."*** *That's a meta-IO. Sub-decompose it: what specifically does the executive need to know, agree to, or approve? Each is a sub-IO. An undiscovered sub-IO here is often the one that blows the schedule at week six.*
> - ***Omitting the `external` IOs.*** *The natural tendency is to only draw what's inside your team's control. But the `external` obstacles are the most dangerous ones — they don't self-resolve, they don't show up in sprint planning, and nobody escalates them until the day before the deadline. Naming them in the PRT is what forces the conversation early.*
> - ***Drawing the PRT before the FRT is stable.*** *The PRT derives its injection from the FRT. If the FRT still has open negative branches or unchecked desired effects, the injection it hands you may change — and with it, the obstacles change. Stabilize the FRT first, then draw the PRT. Doing it the other way around creates rework you'll blame on "the analysis process" when the real culprit was sequencing.*
> - ***Conflating obstacle and IO.*** *The obstacle is what is currently preventing you. The IO is the state you need to reach. If they look the same, one of them is wrong. "No training curriculum" is an obstacle. "Training curriculum drafted and approved" is the IO. They are inverses: the obstacle describes the absence; the IO describes the presence. When you can't tell them apart, re-read aloud: "We don't have X" is an obstacle; "We have X" is the IO.*

> ✏️ ***Now you try.*** *Pick a goal you're genuinely blocked on. Open a PRT (`Cmd+K → New diagram… → Prerequisite Tree`), put the goal at the top, and list the obstacles honestly. Under each obstacle, write the Intermediate Objective that overcomes it. Export the plan (`Cmd+K → Export… → Prerequisite plan (CSV)`) and read the dependency order — which IO has to come first?*

🔁 **Chain to next:** the PRT shows *what's in the way*. The TT shows *the concrete actions that dismantle each obstacle, in order*, with explicit preconditions and expected outcomes at each step.

→ Continue to Chapter 8 — Transition Tree

Chapter 8 — Transition Tree

How do we get there?

> 🎯 ***What this process is for*** *A Transition Tree (TT) is the operational plan: a sequenced set of (action, precondition, outcome) triples that take you from current state to the IOs identified in the PRT. It answers: "What exactly does Maria do on Tuesday morning?"*

The premise

The PRT names obstacles and pairs IOs. The TT decomposes each IO into the concrete *actions* that achieve it, each action gated on a *precondition*, each producing an *outcome*.

The triple structure (action + precondition → outcome) is the smallest unit of TOC's implementation grammar. It maps cleanly to: "Given X, do Y, expect Z." Project plans and operational runbooks live in this grammar.

The distinction from the PRT is worth making sharp. A PRT answer is a *state*: "Ticket-type audit complete" is an IO — it says what the world must look like, not how you get there. A TT answer is an *action sequence*: "Maria pulls the last 6 months of escalated tickets on Monday morning, given that the API token is provisioned, and produces a CSV of ~2400 records." The PRT tells you where you're going; the TT tells you how to walk.

This also means the TT is the level at which implementation plans become testable. Each outcome is a specific, observable state; if the outcome hasn't been produced, the step isn't done. A TT built with that discipline is self-auditing: at any point during execution you can look at the canvas, check which outcomes are marked true, and know exactly what's complete, what's runnable next, and what's blocked.

One more thing the triple structure guards against: the assumption collapse. Every project plan carries hidden assumptions about what makes a step possible. When those assumptions are buried in someone's head they fail silently — the action gets "done" but the expected outcome doesn't appear, because the precondition was never actually met. Writing the precondition explicitly onto the canvas surfaces those assumptions before they fail.

The method, neutral of tool

1. **Start from the highest-priority IO** in the PRT (the one with the most downstream dependents, or the one with the longest lead time).
2. **For each IO, write the *outcome*** — what state of the world must exist when the IO is achieved? (Often the IO statement itself, slightly reworded as an outcome.)
3. **Write the *action*** — the verb-phrase describing what someone does. "Audit the last 6 months of escalated tickets." "Schedule a 2-hour rubric-drafting block on the lead's calendar each Tuesday."
4. **Write the *precondition*** — what must be true *before* the action is doable? "Audit-template is available." "Lead's calendar has been audited and 2-hour blocks identified." This often becomes the *outcome* of an earlier step, creating the sequence.
5. **Group action + precondition with an AND junctor** — both are needed to produce the outcome.
6. **Chain outcomes to next-step preconditions.** The outcome of step N becomes (part of) the precondition for step N+1. The TT reads as a directed chain of triples.

7. **Name the actor.** Each action belongs to a specific person. If you can't name one, the step is not yet operational — it's still a plan on paper.
8. **Stop when each step is something a named person can do in a named day.**

The key test for step granularity: can the person doing the step start without asking any questions that aren't already answered by the precondition? If not, the step is still coarse; decompose further until the answer is yes.

The need → action → expected-effect structure

The structural triple on the canvas (precondition + action → outcome) is the *implementation* layer. Below it sits the classical TT's *reasoning* layer — the internal logic that justifies why this action is the right one. That reasoning runs in the opposite direction: **Need → Action → Expected Effect**.

- The **Need** is the requirement the action is in service of. It links this step to the IO or outcome above it. "We need to know which ticket types are highest-impact so we can scope the training curriculum." Without a need, a step is a habit, not a decision.
- The **Action** itself: what you do.
- The **Expected Effect** — what you expect to be true immediately after the action completes, and why. The expected effect is not the same as the outcome: the outcome is the observable state the step produces; the expected effect is the mechanism. "After pulling and categorizing the CSV, we'll have ranked the ticket types by frequency-weighted resolution time — which means the training-scope decision is data-grounded rather than opinion-based."

This three-part reasoning structure — **Need → Action → Expected Effect** — is where reviewers find the real flaws in a plan. "Is that the right action for this need?" and "does that action reliably produce that expected effect?" are the two questions that catch wrong steps before you build the wrong thing.

In TP Studio, the Need and Working Assumption fields on each Action entity in the Inspector carry this reasoning. The **Working Assumption** is the belief that makes the action sufficient: "a 6-month lookback is long enough to represent the ticket-type distribution." Write it down. If the assumption turns out to be wrong mid-execution, you'll know exactly which step to revisit.

The worked example

From Chapter 7: IO `Ticket-type audit complete`. Let's decompose.

`Cmd+K → New diagram…` → select **Transition Tree**. An empty TT canvas opens.

Step 1 — Triple 1

The IO gives you the terminal outcome. Work backward from it.

- **Outcome:** A CSV of ~2400 escalated tickets with type tags and resolution times.
- **Action:** Pull last 6 months of escalated tickets via the helpdesk's API.
- **Precondition:** API token + read-access scoped to escalation tags.

Add three entities. Set the Action to type `action` (cyan stripe — use the Inspector Type grid, or `Cmd+K → Mark entity as Action (TT)`). The precondition and outcome entities stay as type `effect`. Group Action + Precondition as AND (`Cmd+K → Group as AND`, or select both edges and right-click → Group as AND). Connect to Outcome.

In the Action's Inspector, add:

- **Need:** Know the shape of escalations so training scope is data-grounded.
- **Working assumption:** 6 months of data is representative of steady-state escalation patterns; seasonal spikes don't distort the type distribution.

Step 2 — Triple 2

- **Outcome:** A ranked list of ticket types by frequency-weighted resolution time.
- **Action:** Bucket the CSV by type and compute escalation rate per bucket.
- **Precondition:** CSV exported AND a one-page categorization rubric agreed in advance.

Note Triple 2's precondition includes Triple 1's outcome ("CSV exported"). Connect Outcome-1 into Precondition-2's AND group. The tree now has a chain: produce the CSV, then use it.

The second precondition ("categorization rubric agreed") is independent — it's something the team prepares in parallel. Mark it as a separate effect node, AND-grouped with the CSV.

> 💡 ***Practitioner tip:*** *When a precondition needs to be produced by a different work stream, make it an independent entity and mark it* `effect` *. Don't embed it in the action text — that hides the dependency. The explicit node lets you see clearly that two things must converge before this step runs.*

Step 3 — Triple 3

- **Outcome:** Hardest-20% list signed off by the L2 candidates and the lead.
- **Action:** Mark the top 20% by impact and circulate to L2 candidates for sanity-check.
- **Precondition:** Ranked list available AND L2 candidates identified.

Connect Outcome-2 into Precondition-3's AND group. The "L2 candidates identified" node is a separate precondition that may come from a parallel step in another TT decomposing a different IO.

Step 4 — Triple 4

- **Action:** Publish the hardest-20% list as a workspace doc.
- **Precondition:** List signed off.
- **Outcome: IO achieved.** Hardest-20% list defined.

The TT now reads, top to bottom, as four executable steps, each with a precondition that points back to the prior outcome. A reader looking at it knows exactly what to do on Monday.

Verbalizing

`Cmd+K → Start read-through` opens the walkthrough overlay. For a TT the overlay iterates in topological order. Listen for what sounds awkward: an expected effect that doesn't follow from the action, a precondition that sounds downstream rather than upstream, an outcome that's vaguer than the action that produces it.

For Triple 1: *"In order to obtain 'A CSV of ~2400 escalated tickets with type tags and resolution times', do 'Pull last 6 months of escalated tickets via the helpdesk's API' given 'API token + read-access scoped to escalation tags'."* Sounds right.

For Triple 2: *"…given 'CSV exported AND categorization rubric agreed'."* Pause here. Is the rubric agreed *before* the pull, or *before* the bucketing? It should be before the bucketing. Confirm the structure is correct.

The read-through on a TT is checking two things the structural validator can't: that the precondition is genuinely prior to the action (not a simultaneous concern), and that the outcome is genuinely produced by the action (not just correlated with it).

🛠 **How TP Studio helps:** The `complete-step` validator (CLR tier `sufficiency`, TT-only) fires on any Action whose outgoing edge to its Outcome lacks a non-action sibling (the precondition role). It nudges you toward the triple structure rather than letting you draw a chain of bare actions. The `Cmd+K → Add precondition to Action (TT)` command short-circuits the wiring: select a bare Action, run the command, and TP Studio creates the precondition entity and wires it into the same Outcome automatically.

The second example — a very different domain

The support-team worked example is a *data-pipeline* TT: a series of analytical steps where each outcome is an artifact. Here's a second example in a different shape: a **people-process** TT for onboarding an engineer to production access.

The IO (from a hypothetical PRT): *New engineer has a merged PR in production.*

Work backward. The terminal outcome is "engineer's PR merged and deployed." What action produces it? A reviewer approves the PR. What precondition must hold? The engineer has submitted a PR that meets quality bar.

Now the next layer back: the engineer can only submit a PR once they have a working local environment and a starter ticket. That's the precondition for the "submit PR" action. And a working local environment requires the laptop provisioned AND the dev environment documented. And so on.

Triple sequence:

- **Triple A:** Given *laptop provisioned*, do *provision dev environment via the setup script*, obtain *environment boots clean with all test suites green*.
- **Triple B:** Given *environment boots clean* AND *starter ticket assigned*, do *engineer implements the starter ticket*, obtain *code review-ready PR submitted*.
- **Triple C:** Given *PR submitted* AND *reviewer identified*, do *reviewer runs checklist + submits feedback*, obtain *PR approved with no blocking comments*.
- **Triple D:** Given *PR approved*, do *engineer merges and monitors deploy*, obtain *IO achieved: first production merge landed*.

Notice how the actor changes at each step. Triples A and B belong to the engineer; Triple C belongs to the reviewer; Triple D is a handoff back to the engineer. A TT that doesn't name actors becomes a responsibility-free checklist. When each action specifies who does it, execution is unambiguous.

The **working assumption** discipline pays off here too. For Triple C, the working assumption is: "One reviewer pass is sufficient for a starter ticket." That's probably true for a small ticket and an experienced reviewer — but if the starter ticket is unusually complex, this assumption fails and the step needs an extra review loop. Writing the assumption down means a reviewer reading the plan can flag it early, rather than discovering it after the engineer has already waited a week.

> 💡 ***Practitioner tip:*** *When you have a TT where actors change across steps, number each action with the* ***Step #*** *field in the Inspector (1, 2, 3…). The* ***Task tracker CSV*** *export (* `Cmd+K → Export… → Task tracker CSV` *) then walks the actions in Step # order and emits one row each —* `step / action / precondition / outcome / owner / due_date / status / success_criteria` *— a runnable handoff you can paste straight into Jira, Trello, or a spreadsheet without editing.*

Need and working assumption

The action / precondition / outcome triple is the *structural* unit on the canvas. The classical Transition Tree carries a second layer that lives inside the action itself — the reasoning that justifies it. Besides its Step # ordering, a TT **Action** exposes two optional fields in the Inspector: a **Need** ("why is this step needed?") and a **Working assumption** ("the belief that makes this action sufficient"). Read together with the action, these two fields complete the *Need → Action → Expected Effect* reasoning from earlier in the chapter: the **Need** is why the step exists, and the **Working assumption** is the belief that makes the action sufficient to produce its expected effect. That's the discipline that stops a TT from degrading into an unexamined checklist: every step should be able to say *why* it's there and *what it's betting on*.

Both are free text and both are optional; a fresh step shows neither. Fill them when a step's rationale isn't self-evident — especially the working assumption, which is exactly the place a reviewer will push ("are we sure that's enough?") and exactly the kind of belief worth writing down before the rollout proves it right or wrong.

Action eligibility

Once your steps carry entity states (Chapter 3), the TT stops being a static plan and starts telling you *what's runnable right now*. Select an action and the Entity Inspector shows an **eligibility readout** that folds the effective states of that action's preconditions:

Status	Meaning
Eligible (emerald)	Every precondition feeding the action's outcome is `true`. Go.
Blocked (rose)	At least one precondition is `false`. The readout names the offender so you know exactly what's in the way.
Pending (amber)	A precondition is `unknown` or `disputed` — the step isn't blocked, but it isn't confirmed ready either.
n/a	The action has no precondition slot (a bare action with nothing to gate it).

Only the *true* preconditions count: sibling actions and assumptions feeding the same outcome are ignored, and a precondition that derives `true` from propagation (without a manual state) still makes the step eligible. Combined with what-if speculation (`Cmd+K → Speculate: what changes if…`), this answers the live planning question — "if I flip this upstream outcome to done, which steps unblock?" — without editing the saved plan.

Prefer it on the canvas? **Settings → Display → Show action-eligibility badge** mirrors the readout as an at-a-glance pill on each Action node's right edge — emerald `✓` eligible, rose `✗` blocked, amber `…` pending — so you can scan a whole tree for what's runnable without selecting each step. It's off by default (a state-less tree would read all-pending) and reflects speculation live, just like the inspector readout.

Sequencing multiple IOs

A PRT typically produces several IOs, not just one. The question of how to sequence them across multiple TTs — or whether to combine them into one — is worth getting explicit.

Each IO gets its own TT when the work streams are independent: different actors, different timelines, no shared preconditions between streams. You draw three TTs for three IOs, and the relationship between them lives at the PRT level (this IO unlocks that one), not inside the TT.

Combine IOs into one TT when they share structure: a common early precondition, the same actor at certain steps, or an outcome of one IO that's a precondition of another's action. The combined TT makes the shared structure visible; the separated TTs would hide it, requiring someone to mentally re-connect what you've deliberately disconnected.

The practical tell: if you find yourself writing the same precondition into the bottom of two separate TTs, combine them and share the node.

Sidebars

> ***🛠 How TP Studio helps***
>
> - `Cmd+K → New diagram…` → *select* ***Transition Tree*** *to start fresh.*
> - `Cmd+K → Load example…` → *select* ***Transition Tree*** *to load a worked example with the canonical triple structure. Five TT templates also ship in the Templates library (* `Cmd+K → Browse templates…` *): Support triage Transition Tree, Engineer onboarding, Incident response, Feature-flag rollout, Enterprise deal close — each demonstrating a different domain shape.*
> - ***Inspector Type grid*** *— click* `action` *(cyan stripe) to mark an entity as an Action.* `Cmd+K → Mark entity as Action (TT)` *and* `Cmd+K → Mark entity as desired Outcome (TT)` *are the palette shortcuts — the floating selection toolbar surfaces the same verbs (Mark as Action, Mark as Outcome, Add precondition) as buttons above the selected entity.*
> - `Cmd+K → Add precondition to Action (TT)` *— select a bare Action and run this command to auto-create a precondition entity wired into the same Outcome. Faster than building the triple by hand.*
> - `complete-step` ***validator*** *(CLR tier* `sufficiency` *, TT-only) — flags any Action whose outgoing edge to an Outcome has no non-action sibling (unpaired precondition slot).*
> - ***AND-junctor grouping*** *is essential to the triple structure; the gesture is the same as elsewhere (select edges →* `Cmd+K → Group selected edges as AND` *, or right-click → Group as AND).*
> - ***Action Inspector fields*** *—* ***Step #*** *(ordering),* ***Need*** *(why this step exists),* ***Working assumption*** *(the belief that makes the action sufficient).*
> - `tt-action-locus-unset` ***validator*** *— nudges you to set each Action's Locus (control / influence / external) so the plan reads honestly about authority; an action you can only influence is a different plan than one you control.*
> - ***Action-eligibility readout*** *in the Entity Inspector — eligible / blocked / pending / n/a, folded from precondition states. Mirrors as an at-a-glance ✓ / ✗ / … badge via* ***Settings → Display → Show action-eligibility badge****.*
> - ***Speculate: what changes if…*** *(* `Cmd+K → Speculate: what changes if…` *) — flip any precondition state to true/false without committing, and watch eligibility cascade live across the whole tree.*
> - ***Reasoning as narrative export*** *(* `Cmd+K → Export… → Reasoning as narrative (Markdown)` *) — turns the TT into a numbered list of triple sentences ("In order to obtain X, do Y given Z.") that pastes into a project doc or ticket.*
> - ***Start read-through*** *(* `Cmd+K → Start read-through` *) — iterates each structural edge in topological order for verbalisation discipline.*

💡 ***Practitioner tips***

- ***Name the actor*** *in the Action title — "Maria pulls last 6 months of tickets" not "Pull last 6 months of tickets." Anonymous actions don't get done.*
- ***Estimate effort in the description.*** *TT actions vary from "5 minutes" to "2 weeks." A reader needs to know which. The description field is the right place — the Action title stays clean.*
- ***Use Notes for caveats.*** *The TT is the canonical plan; caveats and contingencies live as Notes beside the chain so the canonical reading stays clean.*
- ***Write the working assumption before you're confident the step is right.*** *The temptation is to fill it in retrospectively ("oh yes, we assumed X"). Write it while designing the step — that's when it's most likely to surface a wrong assumption before any work is done.*
- ***When an upstream IO feeds multiple TTs, mark the shared outcome node's state as*** `true` ***as soon as that IO lands.*** *Eligibility propagates immediately, and everyone working the downstream TTs sees the unblock without a coordination meeting.*
- ***Trace backward from the IO to verify the chain.*** *Once you've built a TT forward (action by action), trace it backward: starting from the IO outcome, ask "what's the last step that produces this?" then "what produces that step's precondition?" The chain should be gapless. Any gap is a missing triple.*

⚠ ***Common mistakes***

- ***Skipping preconditions.*** *A TT without preconditions is just a to-do list. The precondition is what makes the dependency structure visible — and what makes the diagram catch the case where two steps look ready but one is actually blocked. The* `complete-step` *validator flags these; treat them as errors, not suggestions.*
- ***Triples too coarse.*** *"Build the rubric" is not a TT triple; it's an IO. The TT decomposes IOs into steps small enough for one person to do in one sitting. "Maria drafts the severity-1 section of the rubric (≤ 2 hours)" is a triple.*
- ***Mixing IO-level language into TT outcomes.*** *An IO says "ticket-type audit complete" — a state. A TT outcome should say "a CSV of 2400 rows with type tags and resolution times exists in the shared drive" — specific, observable, checkable by anyone. If your outcome could have come from the PRT verbatim, it's still coarse.*
- ***Writing the precondition as a future state.*** *The precondition is what's true before the action, not what the action will produce. "API token provisioned" is a precondition; "API token will be provisioned" is a plan, not a precondition. Tense discipline matters.*
- ***Treating the TT as a one-person artifact.*** *If you built the TT alone and then handed it to the team, expect resistance. A TT built with the actors — especially the precondition and working-assumption fields — is a commitment, not an assignment. The build process is part of the alignment.*

> 🛑 ***When to stop***
>
> - *Every leaf action is sized to fit a person-day or less, with a named actor.*
> - *Every action has its precondition AND-grouped explicitly; the `complete-step` validator shows no open warnings.*
> - *The outcome chain is traceable from the first action to the IO it produces, with no gaps.*
> - *Every action has a Working assumption (or you've decided the rationale is genuinely self-evident for that step).*
> - *The reasoning-narrative export reads as a runnable plan that a new team member could execute without additional briefing.*
> - *You've run what-if speculation to confirm that completing each IO's terminal outcome correctly unblocks the next IO's first step.*

> ✏️ ***Now you try.*** *Take one Intermediate Objective from your PRT. Open a TT (`Cmd+K → New diagram… → Transition Tree`) and sequence the actions as Outcome ← Precondition + Action triples, numbering each step. Set an owner on each Action, then run `Export… → Task tracker CSV` and paste it into your tracker. If a step's precondition slot is empty, the `complete-step` warning will say so — what existing condition does that action rely on?*

🔁 **Chain to next:** the TT is the operational plan. The Goal Tree (next chapter) is a *strategic* decomposition — the frame around the entire CRT→TT process when the constraint is the goal itself.

→ Continue to Chapter 9 — Goal Tree

Chapter 9 — Goal Tree

What does success look like?

> 🎯 ***What this process is for*** *A Goal Tree (originally Dettmer's Intermediate Objective Map) decomposes a single Goal into Critical Success Factors (CSFs) and Necessary Conditions (NCs). It answers: "If success means X, what would have to be true?" Often drawn before a CRT to set the frame; sometimes drawn instead of one, when the problem is "we don't know where to start" rather than "we know things are bad."*

The premise

The CRT works bottom-up from symptoms. The Goal Tree works top-down from objective. The two are duals; you can sometimes infer one from the other.

That duality is worth understanding precisely, because it tells you when to reach for each tool. A CRT asks: *given that these painful effects exist, what cause-structure would explain them?* A Goal Tree asks: *given that this objective matters, what conditions must hold for it to be reached?* The first is a diagnosis; the second is a design brief. Both arrive at a structured map of the same system. When a Goal Tree is drawn for an objective whose non-attainment is itself a UDE — "we are not profitable" — the CSFs and NCs that would have to be true for the Goal tend to map onto the same structural territory as the root causes in the CRT. The NC you're missing is often the root cause you found.

That correspondence is not a coincidence; it reflects the same first principle: a system's underperformance traces to a small number of missing or violated conditions. The CRT names what's wrong. The Goal Tree names what right would look like. Done in sequence, one validates the other.

The Goal Tree is the right starting move when:

- The organization is *planning* (annual strategy, new product launch) rather than *diagnosing* (chronic UDEs).
- You have a high-level goal and need to know what success would actually entail.
- You're trying to align stakeholders before the work starts — the Goal Tree is a shared map of what everyone must agree matters.
- A CRT root cause has just been identified and you want to frame the solution space positively: "what must be true in a world where that cause is gone?"

The structure: one Goal at top; 3-5 CSFs in the middle (the must-be conditions for the Goal); 5-15 NCs at the bottom (sub-conditions feeding each CSF). Each layer connects to the next with necessity edges. Every edge in the tree reads the same way: *"In order to achieve [parent], we must have [child]."* That single reading direction — downward-necessity — is the tree's discipline. When an edge won't submit to that reading, it either belongs in a different diagram or the node is at the wrong level of the hierarchy.

The method

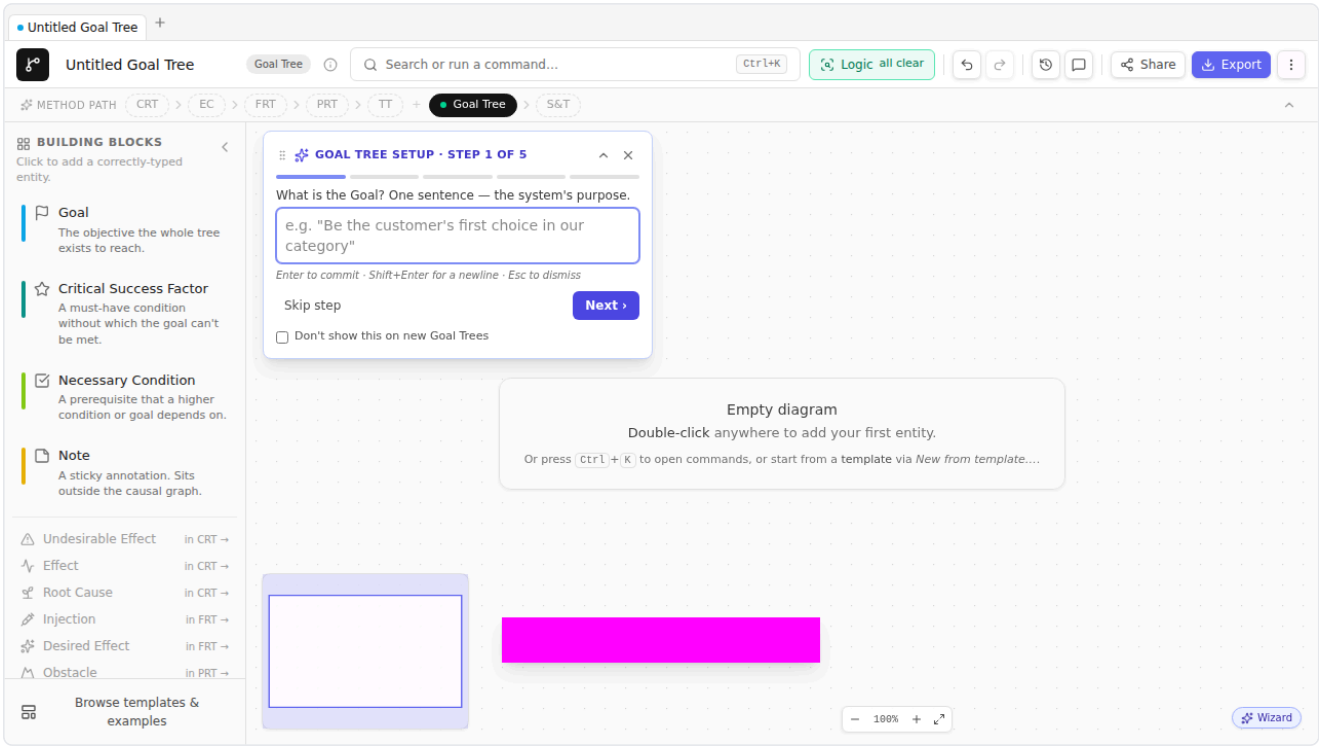
1. **Write the Goal.** One sentence. Specific. Time-bounded if possible. "Hit $10M ARR by EOY 2026." "Ship Customer Portal v2 with 80%+ adoption by Q3." If you can't time-bound it, add at least a measurability criterion: "Hit $10M ARR, measured as MRR × 12 against the last close date." Vague goals produce vague trees.

2. **Brainstorm 3-5 Critical Success Factors.** "What would have to be true for the Goal to be achieved?" Each CSF should be a *condition*, not an action. "Sales pipeline is healthy" is a CSF; "Hire 3 AEs" is not (that's a NC, or a TT action). The CSF layer captures the strategic shape — the few structural realities that are jointly sufficient for the goal. Think of them as the answer to: "What has to be *already in place*, as a state of affairs, for the Goal to be reachable?"
3. **Test necessity at the CSF layer.** For each CSF, ask: "Could the Goal still be reached without this condition?" If yes, the CSF isn't necessary — demote it or reframe it. The CSFs should pass the Dettmer test: remove any one of them, and the Goal becomes unreachable. Add them all together, and the Goal becomes reachable. That's the conjunctive standard.
4. **Under each CSF, list Necessary Conditions.** Sub-conditions that feed the CSF. Aim for 2-4 per CSF. Apply the same necessity test: "If this NC were absent, would the CSF still hold?" If yes, the NC is nice-to-have, not necessary.
5. **Use necessity edges throughout.** "In order to achieve [Goal], we must satisfy [CSF]." "In order to satisfy [CSF], we must have [NC]." The direction is always bottom-up in meaning even when drawn top-down in layout. The tree reads upward — each layer is a *prerequisite* for the layer above.
6. **Walk the tree bottom-up.** Starting from a leaf NC, walk up to the Goal. The chain should read aloud as a coherent argument. "In order to achieve the Goal, we must have CSF-2. In order to have CSF-2, we must have NC-4." If the sentence sounds forced or requires an unstated bridge, there's a missing NC.
7. **Look for missing conditions.** Conjoin all children of a parent. If their conjunction doesn't guarantee the parent, you're missing a NC. That gap is valuable — it names something the planning process hadn't named yet.
8. **Stop when each NC is something you can plan against.** NCs that are themselves wide open (need their own Goal Tree) get bumped to the next iteration; mark them as "needs decomposition" via the Inspector's description field. At the stopping point, every leaf NC should be a condition specific enough that someone could tell you, on any given day, whether it's satisfied.

Worked example — the B2B SaaS GM

Switch gears from the support-team CRT in Chapters 4 and 5. Imagine you're the new GM of a B2B SaaS product line, planning your first year.

`Cmd+K → New diagram…` → select Goal Tree. The **Creation Wizard** opens at step 1, prompting for the Goal.



🛠 **How TP Studio helps:** The Goal Tree creation wizard mirrors the EC wizard's pattern: 5 steps, each prompting for one slot (Goal → CSF 1 → CSF 2 → CSF 3 → first NC). Each step commits live so partial walks leave the canvas in a useful state. The wizard's layout engine places entities in the canonical layered arrangement — Goal at top, CSFs beneath it, NCs at the base — so the structural shape is immediately visible when you dismiss.

Wizard step 1: **Goal**. Type **Hit $10M ARR by EOY 2026**. Next.

Step 2: **CSF 1**. *What's one thing that has to be true for that goal?* **Sales pipeline coverage of 3x quota each quarter.** Next.

Step 3: **CSF 2**. **Net retention >= 110%.** Next.

Step 4: **CSF 3**. **Two new vertical-specific use cases shipped and adopted.** Next.

Step 5: **First NC**. Pick a CSF to decompose. Take "Sales pipeline coverage of 3x quota" — what's required? **AE headcount of 8 by end of Q1.** Commit.

Wizard closes; the canvas now has a Goal Tree with one Goal, three CSFs, and one NC. Continue building manually:

- Under "Sales pipeline coverage", add NCs: **Marketing-qualified-lead flow at 200/mo by Q2**, **Pipeline-review cadence weekly with consistent definition of "qualified"**.
- Under "Net retention >= 110%", add NCs: **Customer success function staffed at 1:1.5M ARR**, **Quarterly business review cadence with strategic accounts**, **Expansion playbook documented + trained**.
- Under "Two new vertical use cases", add NCs: **Product-research effort sized at 1 PM + 1 designer for 8 weeks per vertical**, **2 design-partner contracts signed per vertical**.

You now have a Goal Tree of 1 Goal, 3 CSFs, 7 NCs. Walk it bottom-up and read each chain aloud:

"In order to hit $10M ARR by EOY 2026, we must have sales pipeline coverage of 3x quota each quarter. In order to have sales pipeline coverage of 3x quota, we must have AE headcount of 8 by end of Q1." Coherent.

"In order to have net retention >= 110%, we must have a quarterly business review cadence with strategic accounts." Sounds right — the QBR cadence is what surfaces expansion signals. Mark it.

"In order to have two new vertical use cases shipped and adopted, we must have two design-partner contracts signed per vertical." Sounds right — without committed design partners, there's no feedback loop to validate the vertical use case.

Now apply the missing-condition check. Conjoin all three CSFs: pipeline coverage AND net retention AND vertical use cases shipped. Does that conjunction guarantee $10M ARR? Almost — but there's a gap: what about unit economics? A company can have strong pipeline, strong retention, and new use cases while burning through cash at a rate that makes the revenue number moot. You add a fourth CSF: **Gross margin per customer above 65% at the cohort level**. The tree now has the shape it should.

When the Goal Tree and CRT overlap

You're the same GM. Two months in, you now have a CRT too — drawn from the UDEs the team surfaces in weekly reviews: customers are churning, the sales cycle is elongating, delivery dates are slipping. The CRT resolves to two root causes: no repeatable delivery process, and no structured onboarding playbook.

Look at the Goal Tree. Under "Net retention >= 110%", one of the NCs is "Expansion playbook documented + trained." That NC is exactly what the CRT root cause is pointing at: there is no playbook, and its absence is causing churn. The Goal Tree and the CRT are naming the same gap from opposite directions. That convergence is the strongest possible confirmation that you've found a real structural issue — not an interpretation artifact, but a thing that shows up whether you're reasoning from desired state or from existing symptoms.

When you see this, flag it. In TP Studio: select the NC from the Goal Tree, open the Inspector's description field, and add a cross-reference note to the CRT entity. The formal linking mechanism between diagrams is a cross-doc entity import (`Cmd+K → Import entity from another doc…`) which creates a shadow copy with a back-reference; for a soft annotation, a description note is sufficient. Either way, the convergence becomes visible in a team review.

Worked example II — a product-function transformation

A second example, at a different altitude: a Head of Product who has just taken over a struggling product function. The team ships late, morale is low, and stakeholder confidence is near zero. She needs to align her VP on what recovery looks like before diving into diagnosis.

She opens a new Goal Tree and writes the Goal as: **Product function is a credible delivery and learning system by end of year, as measured by: stakeholders trust our estimates, on-time delivery is >75%, and team NPS >50.**

Three CSFs emerge from her conversation with the VP:

- **Stakeholders trust the team's commitments** — without this, no amount of delivery improvement matters at the political level.
- **The team learns from every cycle** — without this, the function can't self-correct.
- **Work is scoped and sequenced so delivery is predictable** — without this, the other two can't be sustained.

Under "stakeholders trust commitments", she identifies NCs:

- **Estimates are derived from a shared sizing rubric, not gut feel**
- **A delivery date, once committed, is changed by process not by silence**
- **Postmortem findings reach stakeholders, not just the team**

Under "team learns from every cycle":

- **Blameless postmortem within 5 working days of each delivery**
- **Product-metric review cadence monthly, with decisions recorded**
- **Team retrospective bi-weekly with concrete follow-through tracking**

Under "work is scoped and sequenced":

- **Discovery phase explicitly gated from build phase**
- **No more than 3 concurrent discovery-stage projects at any time**
- **Roadmap visible 2 quarters out, with confidence levels marked**

She reads each chain aloud. Most survive. One doesn't: *"In order to have stakeholders trust our commitments, we must have postmortem findings reach stakeholders."* Does that read as necessary? Actually, no — stakeholder trust is built through delivery consistency, not through sharing postmortems. Postmortem visibility is nice but it's not load-bearing. She demotes that NC to a "nice to have" annotation in the Inspector description and removes the necessity edge.

The resulting tree has 1 Goal, 3 CSFs, 8 NCs. She exports the reasoning narrative (`Cmd+K → Export… → Reasoning as narrative (Markdown)`) and sends it to her VP before the alignment meeting. The VP reads it in two minutes and has substantive disagreements about two of the NCs — exactly the kind of early feedback the Goal Tree is designed to surface. The alignment meeting becomes a 30-minute refinement session rather than a two-hour argument about priorities.

Multi-goal Goal Trees

Sometimes the strategic frame has two or three top-level goals. Conventional Goal Tree practice says no — the Goal Tree's discipline is its singular goal. But organizations do sometimes have legitimate dual top-level objectives ("Hit $10M ARR AND keep team headcount under 60").

The multi-goal problem is almost always a framing problem in disguise. The second "goal" is usually one of three things: a constraint on the first goal (headcount cap), a CSF of a higher-level goal you haven't named yet (a board-level objective that includes both ARR and efficiency), or a different planning horizon (the $10M goal is a 12-month target; the headcount discipline is a 36-month sustainability condition). In the first two cases, the right move is to re-frame: add the higher-level goal as the apex, or demote the constraint to a CSF.

TP Studio supports this diagnosis with a *soft* warning: the `goalTree-multiple-goals` validator fires (clarity tier) when a Goal Tree has more than one Goal. The warning is dismissible. The warning's one-click action is `Convert extras to CSFs` — which downgrades every Goal except the oldest to a Critical Success Factor. That's usually the right move: the second "Goal" is actually a constraint on the first.

If you genuinely need two Goals, dismiss the warning and proceed. Just be honest about whether the constraint framing fits better — most of the time, re-framing to a single apex goal produces a richer, more coherent tree.

Sidebars

> **🛠 How TP Studio helps**
>
> - `Cmd+K → New diagram…` → select **Goal Tree** → opens the **Creation Wizard** (Goal → CSF1 → CSF2 → CSF3 → first NC, 5 steps).
> - `goal` entity type (sky stripe), `criticalSuccessFactor` (teal stripe), `necessaryCondition` (lime stripe) — Goal-Tree-specific palette types visible in the Inspector's Type grid.
> - `Add NC` verb: select any CSF or NC in a Goal Tree, single-entity toolbar → **Add NC**. Creates a `necessaryCondition` child connected via a necessity edge. The fastest way to extend the tree after the wizard closes.
> - `Mark CSF` verb: select any entity that isn't a `criticalSuccessFactor` or `goal` → single-entity toolbar → **Mark CSF**. Useful when an entity was created via Quick-Capture and arrived as a plain `effect` rather than the intended tier.
> - `Promote to Goal` verb: select any non-goal entity → **Promote to Goal**. Useful when building bottom-up and discovering that one NC is actually the real strategic objective.
> - `goalTree-multiple-goals` validator with the `Convert extras to CSFs` one-click action — fires when more than one `goal` entity exists; demotes all but the oldest.
> - `goalTree-csf-no-ncs` and `goalTree-csf-count` validators — the first flags a CSF with no Necessary Conditions beneath it; the second is a document-level scope guard on Dettmer's 3–5-CSF band (fewer suggests missing make-or-break conditions; more usually means some are really NCs a tier down).
> - **Templates library** (`Cmd+K → Browse templates…`) ships six Goal Tree starters: Goal Tree starter (generic 3-layer), Sustainable product organization, Profitable subscription business, Trustworthy ML system, Effective sales team, and Generic IT-function goals. Each arrives with realistic NC numbers as a calibration anchor.
> - **Load example** (`Cmd+K → Load example…` → Goal Tree) — the canonical 8-entity "Customer-first" example, useful for studying the structural shape before drawing your own.
> - **Reasoning narrative export** (`Cmd+K → Export… → Reasoning as narrative (Markdown)`) — compiles the tree into a top-down necessity argument sentence-by-sentence, suitable for a stakeholder brief or alignment doc.
> - **Method checklist** (Document Inspector) — five Goal Tree steps tracked per document: State the Goal; List 3–5 CSFs; Identify NCs per CSF; Test necessity at every layer; Look for missing conditions. Useful for a team of analysts working the same document across sessions.

💡 ***Practitioner tips***

- ***Time-bound the Goal.*** *"Hit $10M ARR" is weaker than "Hit $10M ARR by EOY 2026." The time bound is what makes the tree falsifiable — and falsifiability is what makes it a planning tool rather than a wish list.*
- ***CSFs are conditions, not projects.*** *"Healthy pipeline" is a condition. "Run the pipeline-improvement project" is not. If your CSF is a project, you've skipped a level: the project produces a condition, and it's the condition you want in the CSF slot.*
- ***Calibrate NC specificity to a planning horizon.*** *NCs should be specific enough that someone can answer "is this satisfied today?" If the answer requires its own Goal Tree to determine, you're one level too coarse. If it requires a 3-minute data pull, you're probably at the right level.*
- ***Goal Tree first, CRT second*** *when you're planning.* ***CRT first, Goal Tree implicit*** *when you're diagnosing existing pain. Both flows are valid; pick based on what the room knows. Starting with a CRT when the organization is in denial about having problems produces useful diagnosis but poor alignment. Starting with a Goal Tree when the symptoms are acute produces a nice vision doc but no traction.*
- ***Use the Goal Tree as a reading test for your CRT root causes.*** *If you've drawn both, overlay them informally: every root cause in the CRT should correspond to a missing or violated NC in the Goal Tree. When a root cause has no corresponding NC, either the Goal Tree is missing a branch or the root cause is a symptom rather than a cause.*
- ***Re-draw the Goal Tree annually.*** *Last year's strategic frame ages out. The CSFs that were true in year one (acquiring initial customers) are different in year three (retaining and expanding them). If the Goal Tree has been static for two years, you've stopped using it.*

⚠ ***Common mistakes***

- ***Too many CSFs.*** *Three is a sweet spot; five is okay; seven means you haven't decided what matters. The CSF layer is the strategic shape of the goal — it should be small enough that every person in the room can hold it in working memory simultaneously. When you feel compelled to add a sixth CSF, ask yourself whether two of the existing ones could be merged or whether one is really an NC of another.*
- ***CSFs and NCs at the wrong level.*** *"Increase MRR 10% MoM" is too granular for a CSF (that's a metric to track, not a structural condition). "Have a pipeline" is too vague for an NC. The calibration rule: a CSF should be specific enough to be falsifiable but abstract enough that you couldn't work on it directly — you work on NCs that feed it. An NC should be specific enough that it has an owner and a measure.*
- ***Confusing Goal Tree with the to-do list.*** *The Goal Tree is what would be true, not what would be done. Actions belong in the PRT or TT downstream. If you find yourself writing "Hire 3 AEs" as a CSF, you've written a task. Reframe: "AE headcount sufficient to cover quota" is the condition; "hire 3 AEs" is how you get there.*
- ***Leaving necessity untested.*** *Every edge should survive the challenge: "Could the parent be achieved without this child?" If you haven't asked the question, you haven't drawn a Goal Tree — you've drawn a mind map with a nice shape. The necessity discipline is the whole point.*
- ***Treating the first draft as done.*** *The real value of the Goal Tree surfaces in the second pass — after you've walked the tree aloud and found the edges that don't sound right, after you've applied the missing-condition check, and ideally after someone who wasn't in the room has read the reasoning narrative and pushed back. First-draft Goal Trees are usually at the right shape but wrong specificity.*

> 🛑 ***When to stop***
>
> - *One Goal, time-bounded, specific, measurable.*
> - *3-5 CSFs, each a condition (not an action), each genuinely necessary.*
> - *2-4 NCs per CSF, each specific enough to have an owner and a check date.*
> - *Necessity tested at every edge — every child passes "is this really necessary for the parent?"*
> - *Missing-condition check applied at every parent — the conjunction of children is sufficient for the parent.*
> - *Read-aloud passes at every chain, without invented bridge-sentences.*
> - *You can describe the strategic frame in one sentence that the Goal-Tree vocabulary makes precise.*

> ✏️ ***Now you try.*** *State a goal for your team in one sentence. Open a Goal Tree (* `Cmd+K → New diagram… → Goal Tree` *) and let the wizard walk you through the apex Goal, three Critical Success Factors, and the Necessary Conditions under each. Keep it to a single apex — if a second goal sneaks in, the* `goalTree-multiple-goals` *warning offers a one-click* ***Convert extras to CSFs****.*

🔁 **Chain to next:** the Goal Tree is the strategic frame. The S&T tree is the *deployment* of that frame across the organization, one operational level at a time.

→ Continue to Chapter 10 — Strategy & Tactics Tree

Chapter 10 — Strategy & Tactics Tree

Deploying operationally

> 🎯 ***What this process is for*** *A Strategy & Tactics Tree (S&T) is the deployment-grade decomposition of a strategy across an organization, from CEO-level down to individual contributor. Each node is a 5-facet card holding Necessary Assumption (why the strategy matters), Strategy (what), Parallel Assumption (why this specific approach), Tactic (how), and Sufficiency Assumption (why that's enough). Used for cross-functional rollouts, big strategic programs, and as the working document for a TOC-style strategic deployment.*

The premise

The Goal Tree is *what success looks like*. The S&T tree is *what each person does on Monday morning when we deploy the strategy across 30 teams*.

S&T came late to the TOC tradition — Goldratt formalized it in the early 2000s — and is the most operationally specific of the thinking processes. The earlier TPs are all diagnostic: CRT finds the problem, EC names the conflict, FRT tests the fix, PRT sequences the move, TT specifies the steps. The S&T is different. Its job is not diagnosis but *argument*. Every node is a self-contained micro-case: this is the situation, this is our chosen response, this is why we chose it over the alternatives, this is what success looks like, this is why we trust that the action delivers it. The tree as a whole is the organization's argument-for-its-own-strategy, spelled out node by node.

That framing has a consequence: the S&T's value is not primarily in drawing it, but in *reading and challenging* it. A well-built S&T tree exposes every implicit bet the strategy rests on — and does so in a form that invites targeted disagreement. When a department head says "I don't agree with this strategy," the S&T converts that vague objection into a pointed question: *which facet, on which node, do you think is wrong?* That conversion — from diffuse resistance to specific, answerable challenge — is most of what the S&T is for.

The facet names that follow are Goldratt's. The one-line characterizations are how this book reads them in practice.

Facet	A reading
Necessary Assumption (NA)	The world-state above us has changed enough that *not* acting at this layer has become unacceptable. Captures the trigger handed down from the parent node — why this level of the strategy tree must exist at all.
Strategy	A statement of the result we commit to deliver at this layer — what good looks like, not how we get there. Phrased as an outcome, not an action.
Parallel Assumption (PA)	Of the strategies that *could* respond to the NA, why this one. Captures the comparative choice — the alternatives considered and what made this one the right pick.
Tactic	The concrete things people will do. Verbs, not aspirations. Granular enough that you can tell whether they happened. In TP Studio, the entity *title* is the tactic.
Sufficiency Assumption (SA)	If we complete the tactic, the strategy is fulfilled. Captures the bet that Tactic → Strategy is a real causal chain, not a hopeful one.

Notice the structure: NA frames *why act here*, Strategy commits *to what*, PA defends *why this way*, Tactic tells *who does what*, and SA asserts *this will work*. Together they trace the full argument from environmental pressure (NA)

through the chosen response (Strategy + Tactic) to the trust that the response actually closes the loop (SA). Every missing facet is a hidden assumption — one the team is making implicitly, and therefore cannot challenge.

A complete S&T tree at organizational scale might be 40-100 nodes, decomposed across 4-6 levels. Few teams ever build one. The skill matters mostly when you're either *running* a TOC-style deployment or *evaluating one someone else built*. For most practitioners the S&T is a reading tool as much as a drawing tool — you sit down with a strategy document and ask, node by node, whether the five facets are actually there, and whether they're any good.

The method, neutral of tool

1. **State the apex strategy.** The top-level node is the entire program's commitment. Fill all five facets before doing anything else. If you can't fill the NA for the apex, you don't yet understand why the program exists.
2. **Name the tactic that achieves it.** The apex's tactic is the highest-level action — the "we will do X" from which everything else decomposes.
3. **Decompose into 3-5 child strategies.** Each child S&T node represents a sub-strategy that *contributes to fulfilling* the parent's tactic. The parent tactic is the child's context; the child's NA captures what it inherits from the parent.
4. **Fill all five facets at every level before going deeper.** Partially-filled nodes are worse than no nodes — they imply completeness they don't deliver. The `st-tactic-assumptions` validator fires on any tactic that lacks all three assumption types.
5. **Continue down until the leaf tactic is operational** — a thing a named team can plan against within their existing decision-making authority.
6. **Audit the completed tree.** Read each chain of argument from root to leaf. Every parent's tactic should be reachable by the children's strategies in combination. If there are gaps — places where the children's combined output doesn't quite cover the parent's tactic — the tree is incomplete, not the children.

Understanding the five facets deeply

The Necessary Assumption — why this matters

The NA is the trigger, the environmental pressure that makes this level of the tree non-optional. At the apex, the NA answers: *what changed in the world, in the market, or in the organization that made this program necessary?* Lower in the tree, the NA inherits from the parent: *given that the parent tactic commits to X, why does this particular sub-strategy have to exist?*

A good NA is falsifiable. "It's important for us to grow" is not an NA — it's a value that's always been true. "Without 30% growth by Q3 next year, we miss Series C metrics, and our runway ends in Q4" is an NA: it could be wrong, you can check it, and if it's wrong, the strategy it anchors has no foundation.

The NA is also the canary for strategic drift. If you complete a tactic and then revisit the NA six months later, has the underlying trigger still held? If not, the tactic may have been appropriate for a world that no longer exists.

The Strategy — what good looks like here

The Strategy names the outcome this level commits to. It is not the tactic (which comes next) and it is not the NA (which explains why). It answers: *if the tactic succeeds, what condition will exist that didn't exist before?*

Write it as a present-tense fact about the future state: "We are the preferred vendor for mid-market healthcare SaaS compliance," not "we will pursue the healthcare vertical." The tactic covers the pursuit; the strategy names the

destination.

The Parallel Assumption — why this path, not another

The PA is the most commonly skipped facet and the most intellectually honest one. It forces you to name the alternatives. A strong PA reads: "Strategy X chosen over Strategy Y because [reason], and over Strategy Z because [reason]." If you have never written that explicitly, you have not seriously considered the alternatives — you've committed to a path by default.

The PA also carries the information that makes the tree revisable. If the assumption under the PA turns out to be wrong — if the reason you chose X over Y no longer holds — you now know exactly which node needs to change, and which alternatives are already documented and ready to reconsider.

The Tactic — what people do on Monday

The tactic is action-shaped and observable. "Establish relationships with healthcare CISOs" is a tactic. "Prioritize healthcare" is not. If you cannot imagine how you would check whether the tactic has happened, it isn't a tactic yet.

At organizational scale, the tactic at one level becomes the *strategy* at the next level down. The decomposition works because each child node asks: *to achieve this parent tactic as its strategy, what does this team specifically do?* That recursive structure is why the S&T can span from CEO level to an individual contributor's sprint backlog without losing the thread.

The Sufficiency Assumption — the bet that connects action to outcome

The SA is where most strategic plans are the weakest. It says: *doing the tactic will produce the strategy outcome*. That is a causal claim. It can be wrong. Write it as an explicit assertion that can be challenged: "Hiring 2 healthcare-specialist account executives will generate 3 design-partner logos within 6 months, which will generate 12 paid logos within 18 months, sufficient to hit the vertical revenue target." Now you know the mechanism you're betting on, and you can check it.

If the SA turns out to be wrong — the design partners don't convert — you haven't just learned "the strategy didn't work." You've learned *which facet was wrong*, and the tree points you to the NA and PA of the child node responsible, which in turn tells you what the alternative tactic should be.

The worked example

We'll build a 2-level S&T tree for a fictional but realistic situation: **a $10M ARR B2B SaaS company deploying a focused vertical-entry strategy for the next 18 months**.

The setup: the board and CEO have agreed that scattered horizontal GTM is preventing the company from winning competitive deals. The decision is to focus on one vertical for an 18-month push, establish reference customers, and build a moat. You're facilitating the S&T build with the leadership team.

Step 1 — Open an S&T diagram

`Cmd+K → New diagram…`. In the picker, select **Strategy & Tactics Tree**. The canvas opens empty with the method checklist visible in the Document Inspector — six steps, all unchecked. Check off the first two (`st.apex` and `st.tactic`) as you complete them.

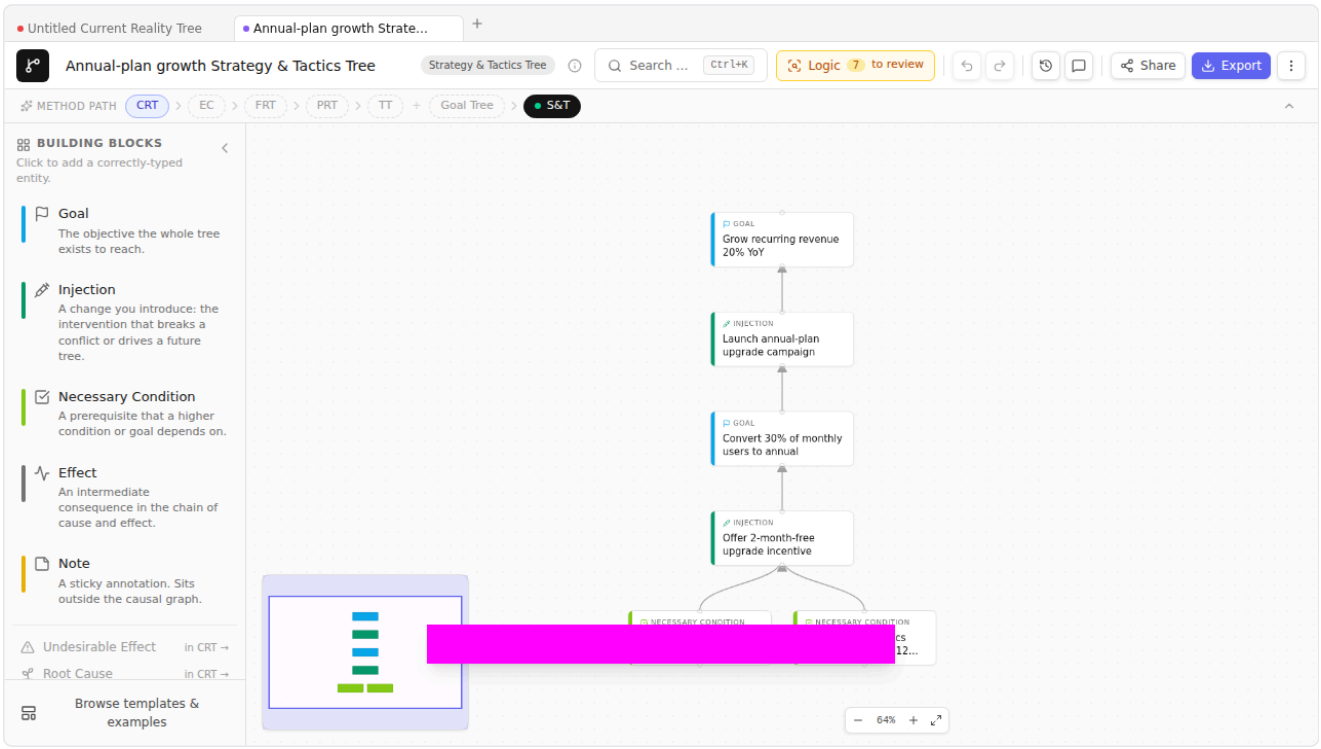
Step 2 — The apex node

Double-click the canvas. Type the tactic for the apex: **"Run an 18-month focused vertical go-to-market for the healthcare compliance segment."** Press Enter. The entity is created. In the Inspector's Entity Type grid, it should already be typed as `injection` for an S&T diagram — this is the tactic type.

Now open the **S&T facets** section in the Inspector (it appears automatically for injection entities in an `st` diagram). Fill the four companion facets:

- **Strategy:** "We are the recognized go-to vendor for mid-market healthcare compliance automation."
- **Necessary Assumption:** "Horizontal GTM has limited deal velocity and win rate. Without a focused wedge we cannot win competitive deals against point-solution specialists, and we will miss our Series B metrics by Q4 next year."
- **Parallel Assumption:** "Healthcare chosen over FinTech because the compliance moat (HIPAA, HITRUST) is more durable and the ICP is more willing to pay for integration. EdTech chosen against because procurement cycles are 2x longer. Healthcare chosen over Manufacturing because our existing integrations are 80% already compliant."
- **Sufficiency Assumption:** "3 design-partner healthcare logos → 12 paid within 12 months → $2M ARR new vertical revenue, sufficient to justify a dedicated vertical pod and hit Series B metrics."

As soon as you fill in the Strategy facet, the canvas node expands from a single-row entity into the tall **5-row card** with each facet visible as its own labeled row. Click any row directly on the canvas to inline-edit it.



Read the apex node aloud, facet by facet. "Because horizontal GTM is limiting us (NA), we're committing to become the recognized healthcare compliance vendor (Strategy). We chose healthcare over FinTech and EdTech because (PA). We'll execute this by running an 18-month focused vertical GTM push (Tactic). We trust this will work because 3 design partners → 12 paid → $2M ARR (SA)." The argument should sound like something a board member would recognize from the off-site.

Step 3 — Three child strategies

The apex tactic — running the focused GTM — decomposes into three parallel sub-strategies. Each is its own full S&T node. Add each one by double-clicking the canvas, then connect them to the apex with a downward edge (the apex's tactic is each child's parent context).

Child 1: Product readiness for healthcare

- **Tactic:** "Build HIPAA-compliant audit logging and two HITRUST-mapped feature flags by Q2."
- **Strategy:** "Product is healthcare-compliant out of the box — no custom implementation work needed for design partners."
- **NA:** "Healthcare CISOs will not approve a vendor that requires custom compliance work — our current product requires 3-4 weeks of post-sale configuration, which killed two deals last quarter."
- **PA:** "Audit logging + 2 feature flags chosen over a full HITRUST certification (18 months, $300K) because certification can follow design-partner acquisition; it's a prerequisite for scaling, not for closing the first 3 logos."
- **SA:** "When design-partner security reviews begin, our product passes baseline checks without custom work, removing the deal blocker that killed Q3 deals."

Child 2: Sales motion for the vertical

- **Tactic:** "Hire 1 healthcare-specialist AE and retrain 2 existing AEs on healthcare-specific discovery and objection handling."
- **Strategy:** "Sales team can run a healthcare-specific discovery call and close motion without generalist toolkits."
- **NA:** "Current AEs use horizontal messaging; healthcare buyers consistently say they 'didn't feel understood.' Of 6 healthcare POCs last year, 4 stalled at discovery."
- **PA:** "Specialist AE hire chosen over sales training alone because healthcare discovery requires institutional knowledge our team doesn't have. External hire can mentor the two retrained AEs, creating a pod."
- **SA:** "A 3-person specialist pod running vertical discovery will convert 50% of qualified healthcare opportunities to POC, up from 17% today."

Child 3: Reference engine

- **Tactic:** "Sign 3 design-partner healthcare logos at 60% list price; deliver weekly executive sponsors for mutual case studies."
- **Strategy:** "We have 3 named, referenceable healthcare logos within 12 months."
- **NA:** "Healthcare procurement is reference-heavy — the top objection in every deal is 'we've never heard of you.' Without logos, sales cycles are 6+ months regardless of product quality."
- **PA:** "Design partner model chosen over standard discounting because design partners provide structured feedback + case study rights in exchange for pricing, yielding both product intelligence and references simultaneously."
- **SA:** "3 referenceable logos reduce average healthcare sales cycle from 6 months to 3 and convert the 4 currently-stalled opportunities, yielding $1.2M ARR in the first 12 months."

Step 4 — Verbalize and challenge each node

Read each child node back to the team. For each one, the most valuable questions are not "do we agree with the strategy?" but:

- "Is the NA still true?" (Could someone argue that horizontal GTM is actually working and we're over-indexing on a few bad deals?)
- "Is the PA defensible?" (Why not EdTech? Why not certification-first?)

- "Is the SA realistic?" (3 logos in 12 months — what's our assumption about deal cycle time?)

The facet structure converts strategic disagreement from vague ("I'm not sure about healthcare") into specific ("I think the PA for Child 1 is wrong — I believe certification is a prerequisite for closing the first logo, not just scaling"). Specific challenges are answerable. Vague ones aren't.

Step 5 — Check completeness

Open the CLR walkthrough panel. The `st-tactic-assumptions` validator fires on any `injection` entity in an S&T diagram that has fewer than three incoming `necessaryCondition` edges. In the TP Studio S&T model, NA, PA, and SA are all represented as `necessaryCondition` entities feeding into the tactic node. If you've filled the four facets in the Inspector's S&T facets section, TP Studio creates those edges automatically — but if you added assumptions manually as separate entities without using the facets section, check that each tactic has all three.

Check the method checklist in the Document Inspector. Tick off `st.na`, `st.pa`, `st.sa`, and `st.decompose` as you complete them. When all six steps are checked, you're through the prescribed method.

Step 6 — Stop

How do you know the S&T is done?

- Every node has all five facets. No empty rows on any canvas card.
- Leaf tactics are operational: a named team can schedule the work this sprint.
- Each parent's tactic is covered by the children's strategies in combination — no coverage gap.
- You've challenged at least the PA on every node. If you haven't, you haven't used the tree.
- You can hand the tree to a deployment lead and they build a rollout calendar from it without structural questions.

S&T vs. Goal Tree vs. project plan

Three tools that can look similar from a distance but do different jobs:

Goal Tree (Dettmer's IO Map): a *necessity* tree that answers "what conditions must all simultaneously hold for the goal to be met?" It is structural and declarative — no assumptions about how conditions are achieved, no sequencing, no agent assignment. Its logic is AND-logic: all the CSFs must hold. Use it to define and communicate *what success looks like in totality*.

S&T Tree: an *argument* tree that answers "what is the organization's case for its own strategy?" Each node carries not just a what (Strategy) and a how (Tactic) but three explicit assumptions that constitute the argument. Use it to deploy a strategy with accountability — and to make disagreement productive.

Project plan / Transition Tree: a *sequence* plan that answers "what do we do in what order, and what must be true to do each step?" Sequencing, dependencies, preconditions. Use it to execute a tactic whose argument is already agreed.

The typical flow in a full TOC deployment: Goal Tree sets the destination → CRT/EC diagnose why you're not there yet → FRT specifies the fix → S&T deploys the fix across the organization → PRT/TT sequence and execute each piece.

Decomposition across organizational levels

The S&T's recursive structure maps naturally onto organizational hierarchy, but the levels are strategic levels, not org-chart levels. The apex strategy belongs to whoever owns the whole program — CEO, VP, program lead. The first decomposition belongs to whoever leads each workstream. The second decomposition belongs to team leads. Leaf tactics belong to named people.

The handoff point between levels is the parent tactic becoming the child strategy. The parent says: "we will run a focused healthcare GTM" (tactic). The child inherits that as its strategy: "given that we're running a focused healthcare GTM, my workstream commits to product readiness" (strategy). The NA at each child level captures what the parent tactic implies for that child: "the focused GTM requires compliant product or deals stall."

This handoff is where strategy most often breaks down in practice. The parent believes the tactic is clear; the child interprets it differently; the children's combined output doesn't add up to the parent's tactic; and nobody notices until the program review six months later. The S&T tree makes the handoff explicit — it's the PA and SA at the parent level that define what the children must collectively deliver, and the NA at each child level that confirms the child understood what was handed down.

> 💡 ***Practitioner tip:*** *when reviewing a multi-level S&T tree for the first time, check the parent's SA against each child's Strategy. If the child's Strategy doesn't obviously contribute to the parent's SA being true, there's a coverage gap — the parent's bet that "doing the tactic will achieve the strategy" isn't supported by the children. This is the most common structural defect in large S&T trees.*

Why facet-level disagreement is productive

Most strategy processes either avoid surfacing disagreement (the HIPPO wins) or surface it too late and too diffusely ("this initiative isn't going well"). The S&T's five-facet structure creates a third option: *early, targeted, answerable disagreement*.

When you and a colleague disagree about a child strategy, the S&T makes you both more precise. Instead of "I don't think this is the right approach," you say: "I accept the NA. I accept the Strategy outcome. I disagree with the PA — I think FinTech compliance has a more durable moat than healthcare compliance, and here's the evidence." That's answerable. You go look at the evidence. You update the PA or you don't. Either way you've had a real conversation about the strategy rather than a political one.

Facet-level disagreements cluster, and the clustering tells you something useful. If everyone agrees with the NA and Strategy but multiple people flag the SA, the team's shared concern is *whether the tactic actually delivers* — a causal chain doubt. If everyone flags the PA, the team's concern is *whether this was the right choice among alternatives* — a comparative judgment. Those are different conversations requiring different evidence, and the S&T lets you have them without conflating them.

Sidebars

> **🛠 How TP Studio helps**
>
> - `Cmd+K → New diagram…` → select **Strategy & Tactics Tree** to start a fresh S&T canvas.
> - `Cmd+K → Load example…` → select **Strategy & Tactics Tree** to load a 2-level reference tree demonstrating the 5-facet rendering.
> - **5-facet card rendering**: an `injection` entity in an `st` diagram with any of the four reserved attribute keys (`stStrategy` / `stNecessaryAssumption` / `stParallelAssumption` / `stSufficiencyAssumption`) filled in renders as a tall 5-row card with labeled rows. Click any row on the canvas to inline-edit it. The entity title is always the Tactic row.
> - **S&T facets section** in the Entity Inspector — surfaces automatically for injection entities in `st` diagrams. The four input fields map to the four reserved attribute keys; filling any one triggers the tall-card layout.
> - `st-tactic-assumptions` **validator** (CLR clarity tier) fires on any `injection` entity in an S&T diagram with fewer than three incoming `necessaryCondition` edges — one for NA, one for PA, one for SA. The warning message names how many facets are missing.
> - `st-tactic-rollup` **validator** (CLR sufficiency tier) fires on a non-apex tactic (`injection`) with no child tactics feeding up into it — a layer that should decompose into the next level down but doesn't. Add its children, or accept it as a genuine leaf.
> - **6-step method checklist** in the Document Inspector (`st.apex` → `st.tactic` → `st.na` → `st.pa` → `st.sa` → `st.decompose`) — tick each step as you complete it. Tracks progress on the prescribed build sequence and surfaces in any PPTX export.

> **💡 Practitioner tips**
>
> - **Fill the PA before you fill the Tactic.** Writing the PA forces you to name the alternatives you rejected, which sharpens the tactic you chose. If you write the Tactic first, the PA tends to become post-hoc justification rather than genuine comparative reasoning.
> - **NA and SA are about argument, not summary.** "NA: It's important to grow" is useless. "NA: Without 30% growth we don't hit Series C metrics, and Series C is needed by Q3 2027" is an NA: it could be wrong, you can check it, and if it's wrong, the node needs to change.
> - **The SA is your public bet.** Write it as a falsifiable causal claim: "doing X will produce Y within Z months because [mechanism]." Now your team — and you, six months from now — can check whether the bet held and why it did or didn't.
> - **A full org-scale S&T is rarely built.** For most purposes the two-level version (apex + 3-5 children) is the sweet spot: it's presentable in an hour, it exposes the major assumptions, and it surfaces the primary decomposition disagreements. Go deeper only when the children's tactics need their own argument defended — when there's genuine ambiguity about whether the child's approach is right.
> - **Read the tree from leaf to apex for a coherence check.** Starting at any leaf, ask: does this tactic's SA contribute to its parent strategy? Does the parent's SA contribute to its parent's strategy? A leaf whose SA doesn't eventually ladder up to the apex strategy outcome is either misplaced or working on the wrong thing.

> ⚠ ***Common mistakes***
>
> - ***Skipping facets to "get to the point."*** *The facets **are** the point. Without them the S&T tree is just a project plan with extra boxes — no assumptions declared, no alternatives documented, no causal claims to challenge. You save 20 minutes of thinking and lose most of the tree's value.*
> - ***Writing vague tactics.*** *"Improve customer relationships" is not a tactic. "Assign a named CSM to each of our top-20 accounts and run a quarterly executive business review" is a tactic. If someone can't tell whether it happened, it isn't a tactic yet.*
> - ***Treating the S&T as a Gantt chart with extra steps.*** *S&T isn't sequencing. It doesn't say A must happen before B. It says B exists to contribute to A's strategy. Sequencing lives in the PRT and TT; the S&T is the argument for **what** those plans must accomplish.*
> - ***Filling the PA with "we considered X but ruled it out."*** *That's the beginning of a PA, not the end. The PA must also say **why** it was ruled out — what assumption, fact, or judgment made X the wrong choice for this moment. Without the "because," the PA can't be challenged.*
> - ***Building the whole tree before challenging any node.*** *An S&T built by one person in isolation is just that person's assumptions, formalized. The value comes from presenting each node — especially the PA and SA — to the people who will execute it, and watching where they push back.*

> 🛑 ***When to stop***
>
> - *Every node has all five facets filled. No empty rows on any canvas card.*
> - *The `st-tactic-assumptions` validator has no open warnings — each tactic has a NA, PA, and SA.*
> - *Leaf tactics are operational: a named team can schedule the work without needing to ask structural questions.*
> - *Each parent's tactic is covered by the combined strategies of its children. No coverage gap.*
> - *Every PA names the alternatives that were rejected and why.*
> - *You can hand the tree to a deployment lead and they build a rollout calendar from it without coming back with structural questions.*

> ✏️ ***Now you try.*** *Take a strategy you're rolling out. Open an S&T tree (`Cmd+K → New diagram… → Strategy & Tactics Tree`), place the apex strategy as a Goal and its tactic below as an Injection, then fill the tactic's **S&T facets** in the inspector — Necessary Assumption, Strategy, Parallel Assumption, Sufficiency Assumption. Writing the Parallel Assumption (the alternatives you rejected) **before** the Tactic is the move that sharpens the choice.*

🔁 **Chain to next:** the seven structured TPs (CRT, EC, FRT, PRT, TT, Goal Tree, S&T) are the canonical kit. The freeform diagram is for *when the structure doesn't fit*.

→ Continue to Chapter 11 — Freeform diagrams

Chapter 11 — Freeform diagrams

When none of the above fits

> 🎯 ***What this process is for*** *A Freeform diagram is TP Studio's escape hatch. No built-in TOC entity types; no diagram-type-specific CLR validators (the structural checks still run); just the canvas plus whatever custom types and attributes you define. Useful when the situation is causal-but-not-canonical: stakeholder mapping, system architecture, knowledge graphs, an early-stage think where you haven't yet decided which TP it'll become.*

When freeform is honest

- **The structure is causal, but the TOC types don't fit.** "Adoption curve diagram with cohorts" — UDE/effect/root-cause doesn't apply. Freeform with custom entity classes does.
- **You're at the "I don't know what I'm doing yet" stage.** Sometimes you sit down with a messy problem and need to draw things while you figure out *what kind of analysis it'll become*. Freeform lets you start without committing.
- **The diagram is a deliverable, not an analysis.** Documentation. A reference model. A team's shared mental model. Freeform gives you the same canvas affordances without prescribing methodology.

When freeform is dishonest

- **You're avoiding the discipline of a real TP.** "I'll just use freeform for now" is sometimes a flag that you haven't decided what you're really doing, and the cost of that ambiguity is high. A CRT *forces* you to find root causes; freeform lets you avoid them. If you find yourself drawing freeform when one of the structured types would apply, that's a signal.
- **You want to skip the CLR.** Only the structural CLR rules fire on freeform diagrams (clarity, entity-existence, causality-existence, tautology, indirect-effect) — none of the type-pattern rules. That's a feature for genuine non-causal diagrams; a bug for "I want my CRT but without the warnings."

Custom entity classes

The real power of freeform is **custom entity classes** — per-document user-defined types. Open the Document Inspector → Custom entity classes section. Click "Add class".

For each class, set:

- **Label.** What it's called. "Stakeholder", "System Boundary", "Risk", whatever.
- **Color.** From a curated palette.
- **Icon.** One of 57 Lucide icons.
- `supersetOf`**.** Optional. Pick a built-in type the custom class *extends* — `effect`, `rootCause`, etc. Validators that fire on the built-in will also fire on your custom class. Useful when you want a renamed type ("Friction Point" superseding `effect`) but want the validators to still apply.

Custom classes live on the document, so they travel with JSON exports + share links. The class palette appears in the Entity Inspector's Type grid alongside the built-ins.

Custom attributes

Each entity carries a `Entity.attributes` record — typed key/value pairs (`string` / `int` / `real` / `bool`). The Entity Inspector exposes a key/value editor below the warnings list. Use it when an entity needs structured metadata beyond the standard fields:

- A stakeholder entity with `decisionAuthority: int`, `name: string`, `lastEngaged: string`.
- A risk entity with `likelihood: real`, `impact: real`, `mitigation: string`.
- A system component entity with `slaTarget: int`, `currentMTBF: int`.

Attributes round-trip through JSON, CSV exports, and OPML exports.

Sidebars

> **🛠 *How TP Studio helps***
>
> - *`Cmd+K → New diagram… → Freeform`.*
> - ***Custom entity classes** + **icon picker** (57 Lucide icons) in the Document Inspector.*
> - ***Per-entity attributes** key/value editor in the Entity Inspector.*
> - ***No diagram-type-specific CLR firing** — only the universal structural rules (clarity, entity-existence, causality-existence, tautology, indirect-effect) apply; no type-pattern rules.*
> - ***Group presets** still work — you can structure regions of a freeform diagram with Negative Branch / Reinforcing Loop / Archive presets.*

> **💡 *Practitioner tips***
>
> - ***Use freeform sparingly.** If 30% of your TP Studio docs are freeform, look at whether you'd benefit from being more disciplined about TP selection.*
> - ***Promote when you're ready.** If a freeform doc starts to look like a CRT-in-progress, switch the diagram type (Document Inspector → Diagram type). Built-in types and validators activate. The structure you already drew survives.*

> **⚠ *Common mistakes***
>
> - ***Using freeform as "default" because you didn't read this book.** The structured TPs exist because they catch errors freeform won't. Reach for them.*
> - ***Inventing 12 custom entity classes for a 20-entity doc.** Custom classes are powerful; they're also confusing for readers. Three classes is a lot.*

> 🛑 ***When to stop***
>
> - *The diagram answers the question it was drawn to answer.*
> - *You haven't accumulated CLR-style mistakes (freeform won't catch them; you have to catch them yourself).*
> - *The set of custom entity classes is small and understandable to a future reader.*

> ✏️ ***Now you try.*** *Find something that isn't a TOC shape — a decision record, an argument you're having, a dependency sketch. Open a Freeform diagram (`Cmd+K → New diagram… → Freeform`) and, in the Document Inspector, define a couple of **custom entity classes** that fit your domain (e.g. Claim, Evidence, Risk). Map the thing. Notice which CLR rules still fire (the structural ones) and which stay quiet — that's the method getting out of your way.*

🔁 **Chain to next:** Part 2 is done — you know every TP and when to use each. Part 3 covers the cross-cutting skills: groups, the CLR in depth, iteration via revisions and side-by-side compare.

→ Continue to Chapter 12 — Groups, assumptions, injections

Chapter 12 — Groups, assumptions, injections

> *Three cross-cutting features. Each is a small addition to your repertoire, but each one shows up in nearly every Part 2 chapter, so this chapter is the canonical reference.*

Groups

A **group** in TP Studio is a labeled rectangle around a set of entities. It's structural metadata (which entities belong together) plus a visual chrome (the boundary box on the canvas). Use groups when a region of your diagram has a coherent meaning that the entities alone don't convey.

Three operations:

- **Create a group:** Multi-select entities (shift-click or marquee), then `Cmd+K → Group selected entities`. Rename it from the Group Inspector.
- **Nest groups:** A group's Group Inspector exposes a "Nest into parent group" picker. Groups can hold sub-groups indefinitely.
- **Collapse a group:** Click the chevron on the group title bar OR Group Inspector → Collapse. The group's contents disappear; the group renders as a single "collapsed-root card" the user can re-expand.

Group presets are the most useful feature here. TP Studio ships five preset (title, color) pairs derived from canonical TOC group naming:

Preset	Color	Use when…
Negative Branch	Rose	The grouped entities are a NB sub-tree (FRT-specific).
Positive Reinforcing Loop	Emerald	The grouped entities form a reinforcing loop — pair with back-edge tagging on the closing edge.
Archive	Slate (collapsed by default)	The grouped entities have been trimmed out of the active analysis but you want to preserve them.
Step	Indigo	Wrapping one (Action + Precondition → Outcome) triple as a unit in a Transition Tree.
NSP Block	Amber	An S&T tree's Necessary condition / Sufficient action / Parallel assumption triple.

Use them. They keep your diagrams legible to other TOC practitioners who read your work.

🛠 `Cmd+K → Move selection to Archive group` is a one-shot that finds the existing Archive group or creates one (with the Archive preset) and moves the selected entities into it. Useful for cleaning a working diagram before presenting. Any existing group can also be **archived in place** — Group Inspector → *Archive (preserve, hide)*, or palette → *Archive / unarchive selected group* — which dims it out of the way; *Show / hide archived groups* brings archived groups back when you need to re-read the path-not-taken.

Assumptions

Assumptions are **edge annotations** — records attached to the specific arrow they pertain to, sitting beside the diagram rather than on the causal path. They are deliberately *not* an entity type: you won't find "Assumption" in any

Type grid. You add one from the **Assumption Well** in the Edge Inspector (or press `A` with the edge selected), and it renders as a violet annotation card near its edge, with a faint dashed line tying the card to the arrow it challenges. Double-click the card (or use the Well) to edit it. The Well works the same on every diagram type — EC, CRT, FRT, all of them. Each assumption record carries:

- A title (the claim).
- A `status` you pick directly: `unexamined`, `valid`, `invalid`, or `challengeable`.
- A free-text rationale (description).
- A link to a related injection (when valid: false → "and here's the fix").
- An `implemented` flag (used in the InjectionWorkbench for FRT carry-forward).
- A **kind** sub-type — `Necessary`, `Parallel`, `Sufficient`, or untyped. A compact chip cycles through the three roles plus untyped, mirroring the S&T facet vocabulary (Chapter 10): a *necessary* assumption is the trigger that makes acting unavoidable, a *parallel* assumption is why *this* approach over the alternatives, a *sufficient* assumption is the bet that the chosen tactic actually delivers. Tagging assumptions by kind lets an S&T reviewer see at a glance which part of each micro-argument an assumption is propping up.

The status field is what turns the assumption list from a brainstorm into a decision record. *Unexamined* = "we haven't decided yet"; *valid* = "we've concluded this assumption holds"; *invalid* = "we've concluded this assumption is wrong" → this is where the cloud evaporates; *challengeable* = "worth attacking" → it lights up the Injection Workbench.

🛠 **Press** `A` with an edge selected to add an assumption to that edge.

Injections

An injection is an entity of type `Injection`. It represents a proposed change — a thing you would *do* to the system. Injections show up:

- In FRTs as the bottom-of-tree entities driving the desired-effect chain.
- In ECs as the resolution to a broken assumption (linked from the Assumption Well).
- In TT and PRT as the top-of-tree thing being decomposed.

Two Inspector flags worth knowing:

- `implemented`: a per-injection toggle. The **InjectionWorkbench** lists all injections in a doc; toggling `implemented` marks one as "done", visually distinct. Useful for tracking rollout progress against an FRT.
- **Linked assumption**: when an injection was drafted as a response to an invalid assumption, the link is stored explicitly so the Assumption Well and the InjectionWorkbench cross-reference.

The injection flower

In Cohen's *TP Basics* an injection is never vetted from one angle. You probe it from three sides: the **Desired effects** it should produce (a Future Reality Tree), the **Negative branch** it might trigger (an NBR), and the **Plan** to implement it (a Prerequisite Tree). In TP Studio those three live as separate documents, stitched together with the "Link to entity in another tab…" cross-document links you met above — which is faithful to the method but scatters one injection's vetting across tabs, where it's easy to lose track of which side you haven't done yet.

"View the injection flower" gathers it back up. From an injection's inspector (or the palette command "View the injection flower…") it pulls that injection's cross-document links into the three petals — Desired effects, Negative branch, Plan — plus an "Other links" catch-all, and reads its development at a glance: the header says "N of 3 sides

developed," and any side you've left empty shows a prompt rather than a blank ("No negative branch linked yet — ask 'what could go wrong?'"). Each row jumps to the linked entity, so the flower doubles as a launchpad back into whichever tab needs more work. It's the one view that answers "is this injection actually finished?" without your having to remember where you put the pieces.

Sidebars

> 🛠 ***How TP Studio helps***
>
> - ***Group presets*** *(* `Cmd+K → Group inspector → Preset` *).*
> - ***Assumption Well*** *in the Edge Inspector (every diagram type) — per-edge assumption records with status + injection links.*
> - ***InjectionWorkbench*** *in the EC Inspector — listing + status of all injections in the doc.*
> - `View the injection flower` *— gathers one injection's cross-document links into Desired effects / Negative branch / Plan petals (+ Other links) and shows "N of 3 sides developed."*
> - `A` ***shortcut*** *with an edge selected — adds an assumption.*

> 💡 ***Practitioner tips***
>
> - ***Use the Archive group preset early.*** *When you're refining a CRT, you'll discover entities that are wrong or redundant. Don't delete them — Archive them. You'll often want to revisit "why did I think this?" later.*
> - ***Surface assumptions before they're broken.*** *The most useful assumptions are the ones you don't yet know are false. List liberally; classify later.*

> ⚠ ***Common mistakes***
>
> - ***Treating groups as visual chrome only.*** *Groups are structural — they affect exports, copy-paste, and the hoist-into-group feature. Use them for meaningful clustering.*
> - ***Letting assumptions accumulate without classification.*** *A list of 30 open assumptions is no better than no list. Classify them. Or remove them.*

🔁 **Chain to next:** the CLR is the discipline that makes assumptions into real reservations. Next chapter goes deep.

→ Continue to Chapter 13 — The CLR

Chapter 13 — The CLR — your validation conscience

> *The Categories of Legitimate Reservation are the discipline checks Goldratt taught for evaluating a causal claim. TP Studio surfaces them automatically as warnings. They are reservations a thoughtful colleague would raise. They are not errors.*

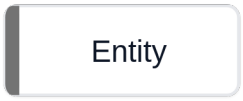
The classical map

Before diving into how TP Studio implements them, here is the classical reference — the eight-box layout Dettmer's *The Logical Thinking Process* uses to teach the CLR, each box pairing the category's question with a tiny example diagram. Cards mirror TP Studio's canvas: amber stripe for causal nodes, neutral grey for effects, red-dashed for the missing element each category catches.

CLARITY

Seeking to understand

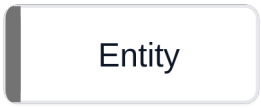
- Would I add any verbal explanation if reading the tree to someone else?
- Is the meaning of words unambiguous?
- Is the connection between cause and effect convincing at face value?
- Are intermediate steps missing (long arrow)?



ENTITY EXISTENCE

Complete, properly structured, valid statements

- Is it a complete sentence?
- Does it make sense?
- Free of embedded "if-then" statements?
- Does it convey only one idea?
- Does it exist in someone's reality?
- Can it be documented with evidence?



CAUSALITY EXISTENCE

Logical connection between cause and effect

- Does an "if-then" connection really exist?
- Does the proposed cause, in fact, result in the stated effect?
- Does it make sense when read aloud exactly as written?
- Is the cause intangible? (if so, look for a confirming predicted effect)

```
marker-end="url(#arrow-indigo)"/> <g filter="url(#card-shadow)"> <rect x="140" y="9" width="100" height="38" rx="5" ry="5" fill="#ffffff" stroke="#e5e7eb" stroke-width="1.25" /> <rect x="140" y="9" width="6" height="38" rx="5" ry="5" fill="#737373"/> <rect x="143" y="9" width="3" height="38" fill="#737373"/> <text x="193" y="33" text-anchor="middle" font-size="13" font-weight="500" fill="#111827" font-family="-apple-system, Segoe UI, Roboto, sans-serif">Effect</text>
```

CAUSE INSUFFICIENCY

A non-trivial dependent element missing

- Can the cause, as stated, result in the effect on its own?
- Are any significant causal factors missing?
- Is an ellipse (AND) required?
- Are any non-dependent causes included by mistake?

```
<text x="60" y="84" text-anchor font-size=" font-weight fill=" font-family apple-system Segoe UI, Roboto sans-serif"
```

```
marker-end="url(#arrow-indigo)"/> <line x1="114" y1="76" x2="158" y2="58" stroke="#dc2626" stroke-width="2" stroke-dasharray="5 3" marker-end="url(#arrow-red)"/> <g filter="url(#card-shadow)"> <rect x="160" y="33" width="100" height="38" rx="5" ry="5" fill="#ffffff" stroke="#e5e7eb" stroke-width="1.25" /> <rect x="160" y="33" width="6" height="38" rx="5" ry="5" fill="#737373"/> <rect x="163" y="33" width="3" height="38" fill="#737373"/> <text x="213" y="57" text-anchor="middle" font-size="13" font-weight="500" fill="#111827" font-family="-apple-system, Segoe UI, Roboto, sans-serif">Effect</text>
```

ADDITIONAL CAUSE

A separate, independent cause producing the same effect

- Is there anything else that might cause the same effect on its own?
- If the stated cause is eliminated, will the effect be (almost completely) eliminated?

CAUSE-EFFECT REVERSAL

Effect misstated as the cause

- Is the stated effect really the cause, and the stated cause really the effect?
- Is the stated cause really a reason why, or just how we know the effect exists?

```
marker-end="url(#arrow-indigo)"/>
<line x1="114" y1="76" x2="158"
y2="58" stroke="#6366f1" stroke-
width="2" marker-end="url(#arrow-
indigo)"/> <g filter="url(#card-
shadow)"> <rect x="160" y="33"
width="100" height="38" rx="5" ry="5"
fill="#ffffff" stroke="#e5e7eb"
stroke-width="1.25" /> <rect x="160"
y="33" width="6" height="38" rx="5"
ry="5" fill="#737373"/> <rect x="163"
y="33" width="3" height="38"
fill="#737373"/> <text x="213" y="57"
text-anchor="middle" font-size="13"
font-weight="500" fill="#111827" font-
family="-apple-system, Segoe UI,
Roboto, sans-serif">Effect</text>
```



```
marker-end="url(#arrow-indigo)"/> <g
filter="url(#card-shadow)"> <rect
x="140" y="9" width="100" height="38"
rx="5" ry="5" fill="#ffffff"
stroke="#e5e7eb" stroke-width="1.25" />
<rect x="140" y="9" width="6"
height="38" rx="5" ry="5"
fill="#737373"/> <rect x="143" y="9"
width="3" height="38" fill="#737373"/>
<text x="193" y="33" text-
anchor="middle" font-size="13" font-
weight="500" fill="#111827" font-
family="-apple-system, Segoe UI, Roboto,
sans-serif">Effect</text>
```

PREDICTED EFFECT

Additional corroborating effect resulting from the cause

- Is the cause intangible?
- Do other unavoidable outcomes of the proposed cause exist besides the stated effect?

```
marker-
end="url(#arrow-
indigo)"/> <line
x1="114" y1="58"
x2="158" y2="74"
stroke="#6366f1"
stroke-width="2"
stroke-dasharray="5
3" marker-
end="url(#arrow-
indigo)"/> <g
filter="url(#card-
shadow)"> <rect
x="160" y="6"
width="100"
height="38" rx="5"
ry="5"
fill="#ffffff"
stroke="#e5e7eb"
stroke-width="1.25"
/> <rect x="160"
y="6" width="6"
height="38" rx="5"
ry="5"
fill="#737373"/>
<rect x="163" y="6"
width="3"
height="38"
fill="#737373"/>
<text x="213" y="30"
text-anchor="middle"
font-size="13" font-
```

```
<text x="210"
      y="84"
text-
anchor="middle"
      font-
size="13" font-
weight="500"
fill="#111827"
      font-
family="-apple-
system, Segoe UI,
Roboto, sans-
serif">New</text>
```

TAUTOLOGY

Circular logic

- Is the cause intangible?
- Is the effect offered as the rationale for the existence of the cause?
- Are other unavoidable outcomes identifiable besides the proposed effect?


```
marker-end="url(#arrow-indigo)"/> <line
x1="134" y1="32" x2="114" y2="32"
stroke="#6366f1" stroke-width="2"
marker-end="url(#arrow-indigo)"/> <g
filter="url(#card-shadow)"> <rect
x="140" y="9" width="100" height="38"
rx="5" ry="5" fill="#ffffff"
stroke="#e5e7eb" stroke-width="1.25" />
<rect x="140" y="9" width="6"
height="38" rx="5" ry="5"
fill="#737373"/> <rect x="143" y="9"
```

```
weight="500"
fill="#111827" font-
family="-apple-
system, Segoe UI,
Roboto, sans-
serif">Effect</text>
```

```
width="3" height="38" fill="#737373"/>
<text x="193" y="33" text-
anchor="middle" font-size="13" font-
weight="500" fill="#111827" font-
family="-apple-system, Segoe UI, Roboto,
sans-serif">Effect</text>
```

This map is the territory; TP Studio's warnings are the tools that walk it. Goldratt's original six are the first six boxes; *Predicted Effect* and *Tautology* are Dettmer's additions, used mainly when a stated cause is intangible (you cannot observe it directly, only its consequences).

The six categories

Goldratt named six. The operationalization into validator-style rules that practitioner tools surface — what TP Studio does — owes a debt to William Dettmer's *The Logical Thinking Process* (2007), which formalized the categories for everyday use. Listed here in order of where they typically bite first as you draft a diagram:

1. **Clarity.** The reader can't tell what you mean. An empty title, a verb-less noun phrase, an abstraction so generic it could be glued to any tree.
2. **Entity existence.** You're asserting a state of the world. Is that state actually observable? "Engineering velocity is dropping" lives or dies on whether you can show it.
3. **Cause-effect existence.** The two entities exist; you've drawn an arrow between them; does the arrow correspond to anything in reality? "Engineering velocity dropped because Mercury went retrograde" fails here, however clean the diagram looks.
4. **Cause sufficiency.** This cause alone — without anything else lined up alongside it — produces this effect? Or does the effect require a co-cause that you haven't drawn? The category where AND-groups get born.
5. **Additional cause.** Is there a *different* cause that would also produce this effect? Two roads to the same destination — modelled with OR-junctors, often missed by single-path thinking.
6. **Cause-effect reversal.** Run the same arrow the other way around; does it still make sense? If the answer is "actually, maybe even more sense," you've inverted the causal direction. The classic trap inside a B2B sales org: "deals are slow because morale is low" — until someone points out the direction is "morale is low because deals are slow."

TP Studio's validator system implements rules drawn from each of these categories. The `Warnings` list in the Inspector surfaces them per-entity / per-edge. The `Start CLR walkthrough` palette command iterates them one at a time.

How TP Studio surfaces them

Each validator carries:

- A **tier**: `clarity`, `existence`, or `sufficiency` — TP Studio's three groupings, not the eight teaching categories above. The tier governs how the Warnings list groups the rule (under **CLARITY** / **EXISTENCE** / **SUFFICIENCY** headers) and the order it appears in the walkthrough wizard.
- A **diagram-type scope**: most rules fire only on specific diagram types (e.g., `ec-missing-conflict` is EC-only; `complete-step` is TT-only).
- A **trigger predicate**: a pure function over the doc that returns the set of entities/edges to fire on.

- Optionally, a **one-click action**: a `WARNING_ACTIONS` registry entry that resolves the warning. Example: the `convert-extra-goals-to-csfs` action on the `goalTree-multiple-goals` warning.

The full list — every implemented rule, its tier, and the diagram types it fires on — is in Appendix C.

Beyond the classical six. Most rules map onto a category Goldratt or Dettmer named. TP Studio adds two families the classical list never did — pure build-quality checks a tool is uniquely placed to compute:

- **CRT build-quality** (CRT only) — *is the tree well-formed?* A UDE with no cause feeding it; a branch that leads to no UDE; a UDE count outside the rough 3–15 band; a leading root cause that explains fewer than half the UDEs; two root causes tied for the lead (a hidden conflict — with a one-click *Spawn Evaporating Cloud*); or a UDE phrased as the absence of a solution.
- **System-dynamics lint** (where loops live) — `logic-type-mismatch` (an edge whose kind fights the diagram's logic), `loop-polarity` (a balancing loop where you'd expect a reinforcing one), `long-arrow` (a jump across ≥ 3 causal levels), and `reinforcing-no-delay` (a feedback loop with no time lag). Note there is *no* cycle warning: loops are auto-detected and the loop-closing edge simply renders as a back-edge — the acknowledged-structure reading is the default, and these lint rules interrogate the loop's dynamics instead.

Each structured diagram type also carries its own shape checks: the PRT's obstacle↔IO pairing rules (`prt-obstacle-no-io`, `prt-io-no-obstacle`), the Goal Tree's CSF coverage and count guards (`goalTree-csf-no-ncs`, `goalTree-csf-count`), the NBR's branch-integrity pair (`nbr-no-negative-branch`, `nbr-ude-disconnected`), the TT's `complete-step` and `tt-action-locus-unset`, and the S&T's facet rules (`st-tactic-assumptions`, `st-tactic-rollup`). Each is introduced in its diagram's Part-2 chapter and catalogued in Appendix C.

None of these is a textbook CLR category — they're the reservations the tool can raise automatically, leaving the contextual judgments to you.

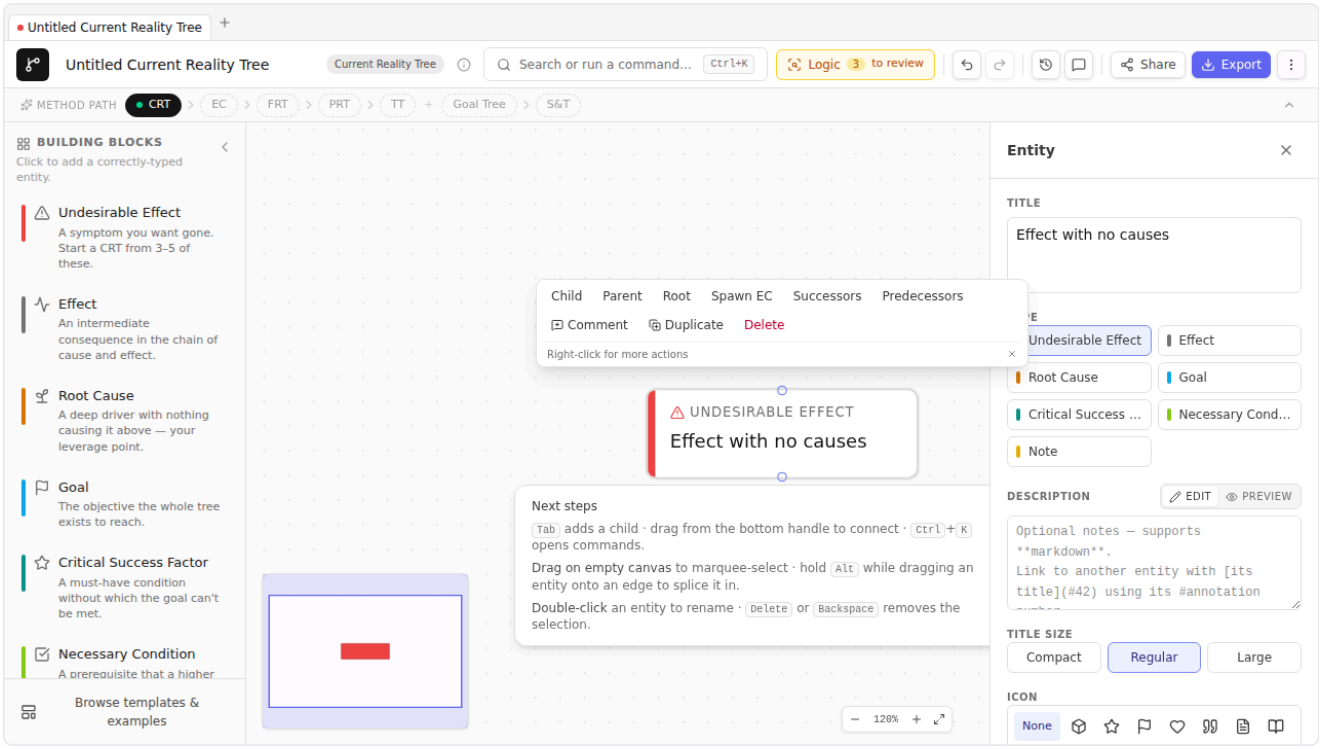
The Logic chip — the whole tree at a glance

You don't have to go hunting for warnings selection by selection. The top bar carries a **Logic chip** that reads the whole diagram continuously: emerald **"Logic · all clear"** when nothing is open, amber **"Logic · N to review"** when reservations remain. Click it and the **Logic check panel** opens down the right side — every open reservation in the document, grouped by tier (**Clarity** → **Existence** → **Sufficiency**), with the open / resolved split in the header. Each row names its target and the rule that fired; clicking a row selects and centres the entity or edge it's about, and the row offers **Resolve / Reopen** in place plus a one-click remedy where the rule has one. A guided-walk stepper at the top pages through the open concerns one at a time.

The same read appears outside the editor: every tree card on the Start page carries a **Logic pill** computed from the identical validation, so triage ("which trees still have logic to resolve?") happens from the workspace — the **Needs review** section is exactly that filter.

Reading warnings

Click any entity with an open warning. The Inspector's Warnings section lists them grouped under their tier — **CLARITY**, **EXISTENCE**, **SUFFICIENCY** — each with a one-line explanation; some carry a "Fix" button when a one-click action is available.



The walkthrough

`Cmd+K → Start CLR walkthrough` opens a modal that iterates every open warning, one at a time, with **Resolve** / **Open in inspector** / **Dismiss** actions. Useful for ratcheting through a complex diagram's warnings without manually clicking each entity.

The walkthrough is scope-limited to *open* warnings — once you dismiss a warning, it doesn't reappear unless the underlying condition changes. That makes the walkthrough a *clearing* gesture: run it before declaring a diagram done; if it's empty, you've considered every reservation.

Scrutinizing a single edge

The walkthrough clears what the validators *fired*. But absence of a warning is not absence of a reservation — a rule fires only when it can detect its trigger condition, and most of the CLR is too contextual for a predicate to catch. An edge can be warning-free and still be sloppy. The complementary move is to take one claim and interrogate it against the whole CLR by hand.

Select an edge and run **"Scrutinize this edge (walk the CLR questions)"** from the palette, or press the **"Scrutinize against the CLR"** button in the edge inspector. Either opens a walk through all eight canonical categories — Clarity, Entity existence, Causality existence, Cause sufficiency, Additional cause, Cause–effect reversal, Predicted-effect existence, Tautology — one at a time, *including the categories nothing flagged*. Each step states the category as a question, shows any warning the validators did raise on that edge, and gives you a checkbox to tick as you genuinely consider it. The button is read-only, so it works under Browse Lock — you can scrutinize a colleague's diagram in a review without touching it. Where the walkthrough is a clearing gesture across the diagram, scrutiny is a *deepening* gesture on one edge: it forces you to ask every reservation of a single claim, not just the ones a rule happened to notice.

The long-arrow check and the missing-step prompt

One failure mode doesn't look like a failure: a clean arrow with no warnings, running across too many causal levels at once. An arrow from "we cut the QA budget" straight to "customers churn" is technically an existence claim — it asserts a cause–effect link — but it's almost certainly hiding one or more unstated intermediate effects. Goldratt called these *long arrows*, and Dettmer flags them as the classic *additional cause / missing step* concern: the stated cause may be necessary, but the path from it to the stated effect passes through at least one step nobody wrote down.

A new EXISTENCE-tier validator flags sufficiency arrows that skip three or more causal levels — measured by the shortest path between the two entities through any already-drawn intermediate nodes. When it fires, the Inspector shows an amber **"Arrow may skip intermediate steps"** warning on the edge. A **"Insert a step"** one-click action is available: it splices a blank intermediate entity into the middle of the over-long edge, wires the two shorter arrows in its place, and leaves the new entity selected and ready to name. You're not told what the missing step is — that's your job — but the seam is opened for you.

For the QA budget example, the check flags the single arrow and you insert the missing middle. A plausible fill: *"Escaped defects rise"* sits between the budget cut and *"customer trust erodes"*, which then connects forward to churn. Two shorter, more defensible arrows replace the one long leap.

Companion check — reinforcing loop with no delay. A second validator in the same tier watches for reinforcing feedback loops (A → B → A cycles, or longer) where no edge in the loop is marked as delayed. A reinforcing loop with no time lag reads as escalating instantly — which is almost never true and almost always a sign that one of the links needs a delay marker. When the check fires it flags the loop with a **"Reinforcing loop — consider adding a delay"** warning and highlights the edges involved. Mark one edge as delayed (the delay toggle is in the edge inspector) to resolve it.

Dismissing warnings

Two ways to dismiss:

- **Resolve** — fix the underlying condition. The warning stops firing on its own.
- **Dismiss with explanation** — keep the condition, but record in the entity's `description` *why* you're accepting the reservation. The walkthrough stops surfacing this one. The audit trail lives in the description.

Don't dismiss without writing the explanation. A dismissed warning with no rationale is a debt you'll pay later when someone reads the diagram and asks "why does this UDE have no causes?"

Sidebars

> ***🛠 How TP Studio helps***
>
> - ***Logic chip*** *in the top bar — the whole-tree open-reservation count, one click from the Logic check panel (grouped by tier, locate-on-canvas, one-click remedies).*
> - ***Per-entity / per-edge Warnings list*** *in the Inspector.*
> - `Cmd+K → Start CLR walkthrough` *— modal that iterates open warnings.*
> - ***One-click actions*** *on a subset of warnings (e.g.,* `convert-extra-goals-to-csfs` *).*
> - ***Tier grouping*** *in the warnings list: warnings sort under* ***CLARITY → EXISTENCE → SUFFICIENCY*** *headers.*
> - ***Dismissibility*** *— every warning can be dismissed; dismissals don't recur until the underlying state changes.*
> - `Scrutinize this edge (walk the CLR questions)` *— walks one edge through all eight CLR categories as questions, including the ones nothing flagged; read-only, so it works under Browse Lock.*
> - ***Long-arrow check*** *— EXISTENCE-tier validator flags sufficiency arrows spanning ≥3 causal levels; "Insert a step" action splices a blank intermediate entity into the over-long edge.*
> - ***Reinforcing-loop delay check*** *— flags reinforcing loops where no edge carries a delay marker.*

> ***💡 Practitioner tips***
>
> - ***Walk-through before you present.*** *A reader will hit the warnings if you don't.*
> - ***Treat warnings as suggestions, not commands.*** *The CLR is a discipline framework; it doesn't always know your context. Apply judgment.*
> - ***The clarity tier is the most forgiving and most pedagogical.*** *If you're learning, work the clarity warnings until they're empty before tackling the higher tiers.*

> ***⚠ Common mistakes***
>
> - ***Dismissing without explanation.*** *Each dismissal is a future-self bug if the rationale isn't recorded.*
> - ***Treating all warnings as equal.*** *The three tiers are deliberately ordered. An* `existence` *warning is structural — a missing entity or an unreal link; a* `clarity` *warning might just be a typo.*

🔁 **Chain to next:** the CLR is the validation conscience. Iteration is the *building* conscience — revisions, branches, side-by-side compare are how a diagram improves over time.

→ Continue to Chapter 14 — Iteration

Chapter 14 — Iteration — revisions, branches, compare

> *A CRT done well is the diagram you have on Friday, not the diagram you drew on Monday. The H1–H4 features make iteration cheap, which is why the diagram you have on Friday is better.*

Capturing snapshots

A **revision** is a point-in-time snapshot of the document. TP Studio stores up to 50 revisions per document in `localStorage`, indexed by id with optional labels.

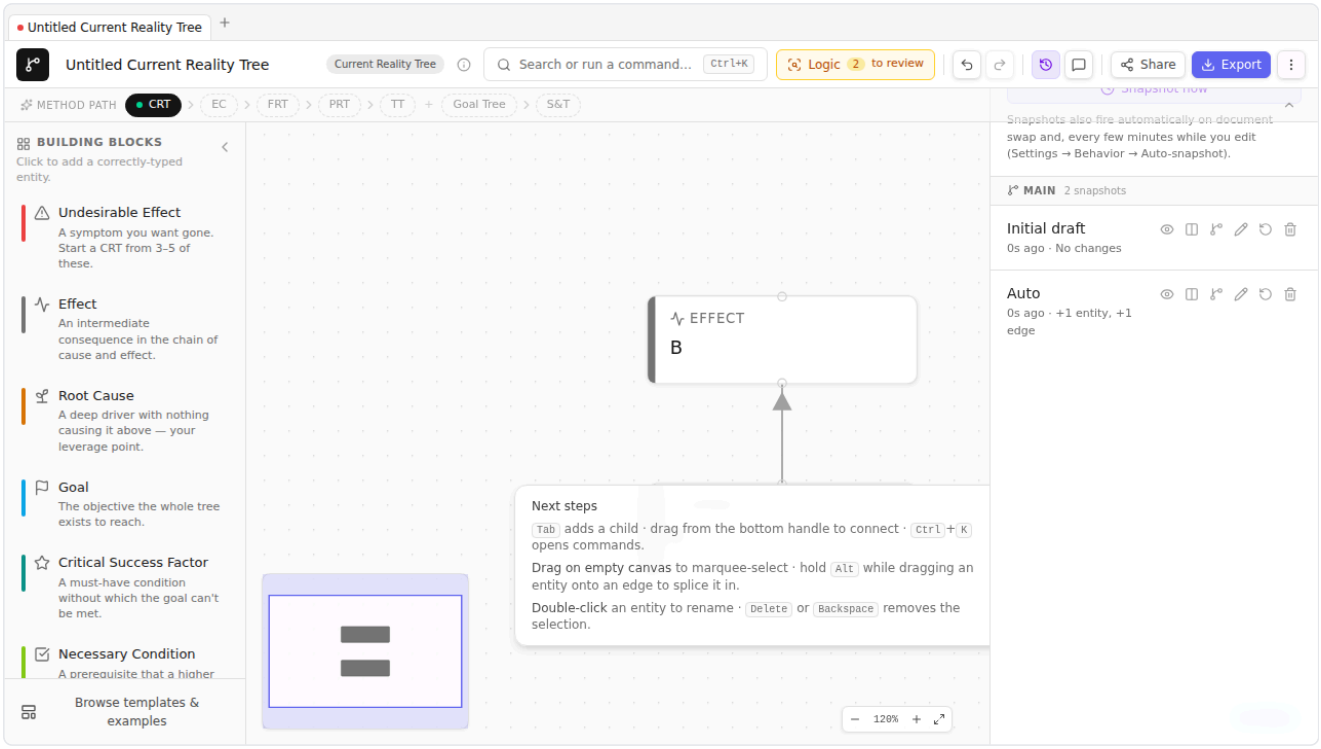
Capture one any time with `Cmd+K → Capture snapshot`. The palette command prompts for a label — give it one. "Before injecting the resolution." "After CRT review with Maria." "Pre-Negative-Branch hunt." The labels are how Future You will find this revision when the History panel has 30 of them.

Automatic snapshots also fire when:

- A document is loaded via Import / Load example / New from template (the prior doc is auto-snapped).
- A "safety snapshot" is captured before a Restore operation (so you can undo the undo).
- **Every few minutes while you edit** — the *Auto-snapshot while editing* setting (Behavior tab, on by default) captures an `Auto`-labelled revision whenever the tree has actually changed since the last one, so a long single-sitting session accrues rollback points without you thinking about it. Turn it off in Settings → Behavior if you prefer manual-only snapshots.

The History Panel

The TopBar's **History** button (it folds into the ⋮ overflow on narrow windows) opens a slide-in panel listing all revisions:



Each row shows the label, the timestamp, and three actions: **Restore** (replace the live doc with this revision), **Compare** (open the side-by-side dialog), and **Branch from here** (give this revision a `branchName`).

The panel groups revisions by branch. The default branch is `Main`. Branching is the way to explore alternatives — "what if we drop the L2 training and just hire one senior?" — without committing your speculation to the canonical history.

Branching

The **Branch from here** action on a revision row in the History panel prompts for a branch name. The branch becomes a labeled lineage; subsequent snapshots after a restore-from-this-branch belong to the new branch.

TP Studio's branch model is intentionally lightweight. It's not git. There's no merge; there's no checkout. A branch is just a label attached to a revision (and its descendants) so the History panel can group them visually. The point is exploration — "I want to keep my current line of analysis but also try this other thing."

🛠 **When to branch:** before a structurally risky edit ("I'm about to delete five entities and re-draw the cause chain"), before exploring a Negative Branch hunt that you might roll back, before presenting to stakeholders (so the live version is "what we showed them" and your subsequent edits don't change it retroactively).

Side-by-side compare

The H4 feature. Select a revision in the History panel → Compare. A fullscreen modal opens with two panels: snapshot on the left, live on the right.

Entities render as absolute-positioned cards; edges as SVG lines between them. Added entities are emerald-tinted; changed entities are amber-tinted; removed entities appear in the snapshot panel only (not in the live one).

Useful when you've done significant edits and want to see *exactly* what changed structurally. Common workflow: capture a snapshot, do 30 minutes of edits, side-by-side compare to verify the diff matches your intent.

Visual diff overlay (compare mode)

Lighter than the side-by-side dialog. The **visual diff** action (the 👁 button) on a revision row overlays a per-entity ring tint on the live canvas:

- Emerald ring = added since this revision.
- Amber ring = changed since this revision.

A banner at the top tells you what revision you're comparing against. Esc exits compare mode.

Useful for "I want to know what's new in the live doc without leaving the canvas." The side-by-side dialog gives you more detail; this overlay is faster.

Sidebars

> ***🛠 How TP Studio helps***
>
> - `Cmd+K → Capture snapshot` *(with optional label).*
> - ***History panel*** *slide-in (the TopBar History button).*
> - ***Auto-snapshot while editing*** *(Settings → Behavior, default on) — periodic* `Auto` *revisions while you work.*
> - ***Branch from here*** *action on any revision row.*
> - ***Side-by-side compare*** *dialog.*
> - ***Compare mode overlay*** *on the live canvas with per-entity diff tinting.*
> - ***Safety snapshot on restore*** *— you can undo a restore.*

> ***💡 Practitioner tips***
>
> - ***Label snapshots when you take them.*** *A timestamp alone is meaningless after a week.*
> - ***Snapshot before risky edits.*** *Cheaper than reconstructing.*
> - ***Branch before exploring.*** *"Try this thing" is much cheaper psychologically when you know you can return to the prior branch.*
> - ***Use side-by-side compare before presenting.*** *Comparing the "what stakeholders saw" snapshot to the live doc surfaces the changes you want to talk about.*
> - ***Capture review feedback as comments, not memory.*** *"After CRT review with Maria" is a set of comments pinned to the edges she questioned — they travel with the file, so the feedback and the diagram never drift apart (see Chapter 16).*

> ***⚠ Common mistakes***
>
> - ***Snapshot drift.*** *Capturing 50 unlabeled snapshots; later not knowing which is which. Label.*
> - ***Treating the history as version control.*** *It's not. There's no commit message, no diff in textual form, no merge. It's a time machine for your canvas, not git.*

🔁 **Chain to next:** Part 3 is done — you have groups, the CLR, and iteration. Part 4 takes the diagram out of your private workspace and into the world: verbalisation, sharing, workshops.

→ Continue to Chapter 15 — Verbalisation and walkthroughs

Chapter 15 — Verbalisation and walkthroughs

> *Verbalisation is the TOC tradition's name for "read it aloud, edge by edge, slowly." It is the single most under-rated discipline in TOC and the source of most of the "wait — actually I had this wrong" moments. TP Studio supports it four ways.*

Why aloud

When you read a diagram silently, your brain auto-completes. The arrow you drew Monday says what you *thought* it said when you drew it. Three days later you skim it and your skimming-brain confirms the original intent — even when the entity titles you wrote on Tuesday subtly broke the logic.

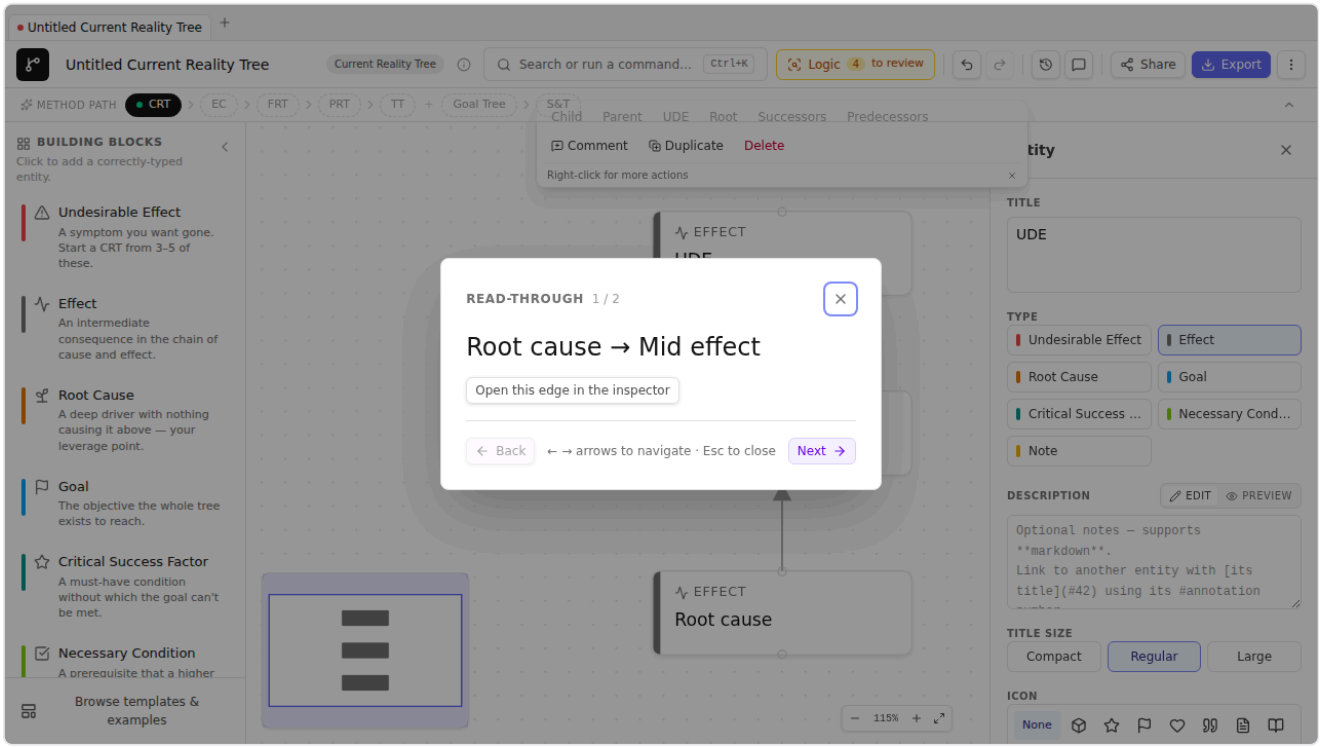
Reading aloud bypasses the autocompletion. The mouth and ear catch what the eye misses. The sentence "Because we ship slowly, customers churn" is structurally fine; the sentence "Because we ship slowly, our quarterly earnings" sounds incomplete in a way you'd notice immediately spoken but not noticed reading.

Goldratt and the practitioners who followed him (Dettmer especially) treat verbalisation as discipline, not affectation. You don't whisper through it; you actually say the sentences. In a workshop, the lead asks one participant to read; everyone else listens.

TP Studio's four supports

1. The Read-through overlay

`Cmd+K → Start read-through` opens a fullscreen overlay that walks every structural edge in topological order. Each step presents one sentence — *"Because [source], [target]"* — with arrow / space navigation between sentences.



Use it solo before you present. Use it with a participant in a workshop. The discipline imposes itself — you can't skim; the overlay shows one sentence at a time.

2. The all-at-once dialog

`Cmd+K → Read entire diagram at once` opens a scrollable panel rendering every edge's sentence in topological order in a single view, with a **Copy all** button that drops the full transcript into the clipboard. Use this when the step-through becomes tedious — a 50+ edge CRT has 50+ clicks in the read-through overlay, but the all-at-once dialog presents the same content in one scroll. Particularly useful for capturing the whole diagram's reasoning into a brief, postmortem, or Slack message without re-typing it.

The two modes are complementary: step-through forces *discipline* (you can't skim, you have to advance), all-at-once gives you *artifact* (the verbal form lives outside the canvas now).

The reading word. Both modes render each edge as a sentence; which connective they use is set by **Settings → Display → Causality reading** — *Auto* (diagram-type-aware, recommended), *Because* / *Therefore* (sufficiency readings), or *In order to* (necessity). A per-edge label always overrides the fallback.

3. The VerbalisationStrip (EC-only)

For Evaporating Clouds, the canonical reading is a single paragraph. The VerbalisationStrip renders it above the canvas, updating live as you edit. The Document Inspector's **EC verbal style** field (per-document) toggles between *Neutral* ("we must") and *Two-sided* ("I want / they want") framings.

Read the strip aloud after every structural edit on an EC. The "this sounds wrong" reaction is the most reliable validator the EC has.

4. The reasoning narrative / outline exports

Two Markdown exports under `Cmd+K → Export → Markup → Reasoning`:

Export	Shape	Use case
Reasoning narrative	Sentence-per-edge, prose paragraphs by chain	Pastes into a doc or a strategy brief; reads as argument prose.
Reasoning outline	Headings = terminal effects; nested bullets = cause chains	Outline form (like OPML); good for review meetings.

Both carry preambles: title, System Scope (if filled), EC conflict statement (on EC docs), and a per-diagram-type appendix (Core Driver for CRTs, triple-form for TTs).

The exports are how verbalisation leaves your screen. The audience that didn't sit through the workshop reads the narrative; they hear what you would have said.

Workshop technique — "read it back to me"

Three-person variant useful for workshop facilitation:

1. **Builder** drives the canvas, says nothing aloud.
2. **Verbaliser** reads each new edge aloud as it's drawn: *"You're claiming because A, B?"*
3. **Listener** says "okay" or "wait — really?"

The verbaliser is the discipline; the listener is the CLR. The builder is the analyst. Switching seats every 15 minutes spreads the discomfort of being verbaliser around the room.

Sidebars

> ***🛠 How TP Studio helps***
>
> - `Cmd+K → Start read-through` *— step-through fullscreen verbalisation overlay (one edge per page).*
> - `Cmd+K → Read entire diagram at once` *— single-view scrollable panel with Copy-all (full transcript).*
> - ***VerbalisationStrip*** *on EC canvas — live paragraph rendering.*
> - `Cmd+K → Start CLR walkthrough` *— partner discipline; iterates open warnings.*
> - ***Reasoning narrative export*** *— Markdown paragraph form per chain.*
> - ***Reasoning outline export*** *— Markdown nested-list form.*
> - ***EC Verbal Style toggle*** *— neutral vs. two-sided framing.*

> ***💡 Practitioner tips***
>
> - ***Aloud means aloud.*** *"I read it in my head" isn't verbalising.*
> - ***Read in the morning.*** *Tired brain skims worse than fresh brain. Don't verbalise after lunch.*
> - ***Capture a snapshot before you re-read.*** *When verbalising surfaces a structural problem, you'll be editing in five minutes. A pre-read snapshot is the baseline you'll compare against later.*

> ***⚠ Common mistakes***
>
> - ***Skipping verbalisation because the diagram "looks right."*** *Looking right and reading right are different. The diagram that looks right but reads wrong is the most common bug.*
> - ***Speed-reading the strip.*** *Slow down. The point of aloud is the time spent on each sentence.*

🔁 **Chain to next:** verbalised diagrams want to be shared. Next chapter covers exports, share links, prints.

→ Continue to Chapter 16 — Sharing your work

Chapter 16 — Sharing your work

> *TP Studio is local-first; nothing leaves your browser unless you choose to send it. The Export picker is the single doorway out. This chapter is the doorway's map.*

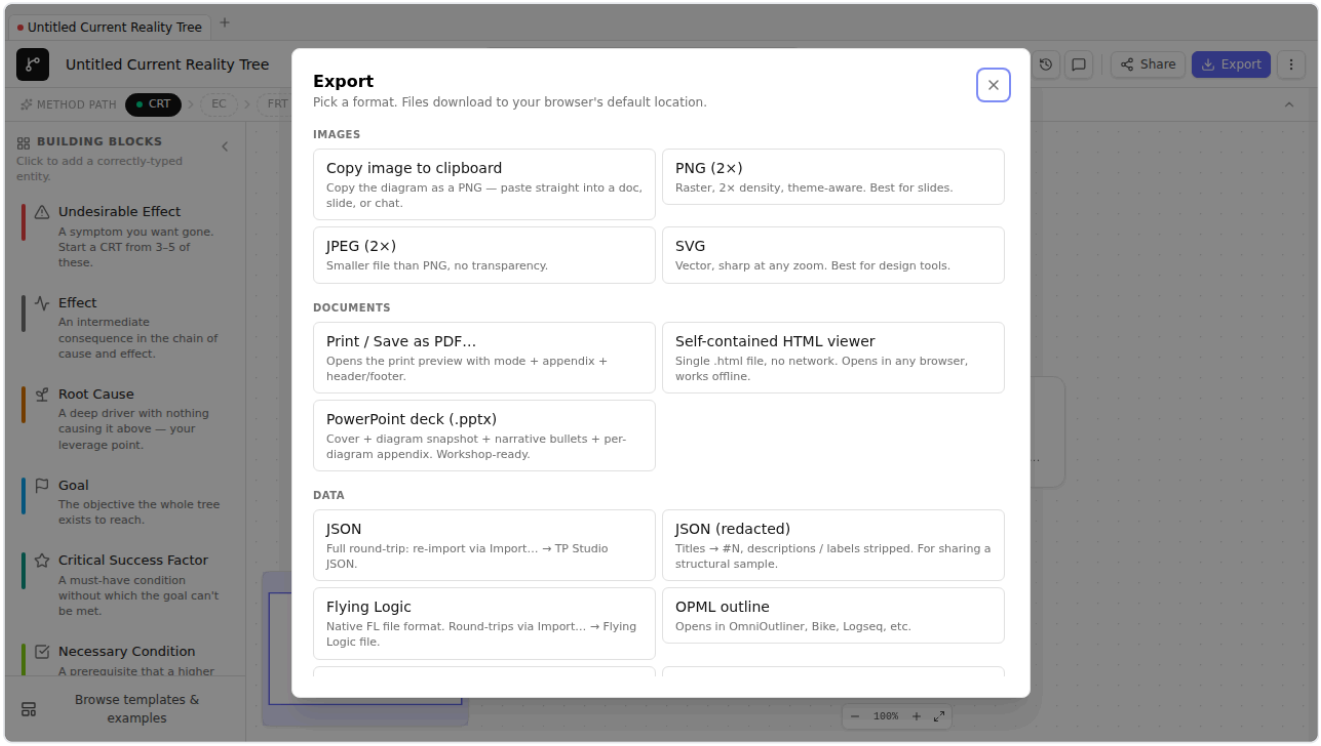
Three modes of sharing

Mode	Best for	Tradeoff
Standalone HTML viewer	Sharing the full diagram with someone who doesn't have TP Studio. Double-click to open.	Single self-contained file. Read-only. Heaviest in size.
Read-only share link	Quick share via Slack / email / chat. URL fragment encodes the full doc.	URL gets long for big diagrams. Receiver opens in their own TP Studio session (Browse Lock auto-engages).
Image / vector export	Pasting into a deck, doc, or wiki.	Static. Lossy if PNG; vector if SVG/PDF. No interactivity.

Pick by the audience. Engineers and analysts: share link. Slide deck and stakeholder pack: vector PDF. The non-technical "open this with the file" audience: standalone HTML viewer.

The Export Picker

`Cmd+K → Export` opens the unified picker:



Five groups — **Images**, **Documents**, **Data**, **Annotations & reasoning**, and **Share** — and the picker remembers your last-used format (reopen it and that card is focused, marked "Last used"):

Images

- **Copy image to clipboard** — puts the diagram on the clipboard as a PNG; paste straight into a doc, slide, or chat.
- **PNG** — high-DPI raster, theme-aware. 2× density by default. Good for ad-hoc paste.
- **JPEG** — lossy raster. Smaller files. Web-friendly.
- **SVG** — vector. Sharp at any zoom. Importable into Figma, Illustrator, etc.

Documents

- **PDF** — true vector PDF via jspdf + svg2pdf. Paginated if the diagram exceeds page-height. Optional annotation appendix, reasoning narrative, and per-type how-to-read legend.
- **Print / Save as PDF…** — opens the **Print Preview Dialog** with mode picker (Standard / Workshop / Ink-saving), page setup (A4 / Letter, orientation, fit page / fit width), annotation appendix + reasoning + legend toggles, and header/footer merge fields. The browser print dialog opens after.
- **PowerPoint deck (.pptx)** — workshop-ready `.pptx` with one slide per major section: cover (doc title + diagram type), System Scope (if filled), an embedded screenshot of the canvas (a tall diagram tiles across several full-width slides instead of shrinking), EC conflict statement (EC-only), paginated reasoning bullets (≤7 sentences/slide in topological order, assumption-edges filtered), Likely Core Driver(s) (CRT-only), and Method-checklist progress. Indigo brand band; vector content where possible. Generated client-side via lazy-loaded `pptxgenjs` — first invocation downloads the vendor chunk (~123 KB gz), subsequent exports are instant.
- **Standalone HTML viewer** — single self-contained file (covered below).
- **EC Workshop Sheet (PDF)** — one-page PPT-style layout with guiding questions, EC-only.

The PDF route is the most polished for static layout. Use Print Preview for the layout knobs, the direct PDF export for unattended use, the PowerPoint deck when you need a slide per section of the analysis for a presentation.

Data

- **JSON** — canonical schema-versioned export; the most lossless format.
- **Redacted JSON** — content-stripped JSON. Same structure, generic titles. Useful for sharing the *shape* of an analysis without the confidential content.
- **CSV** — entities + edges + groups in one RFC-4180 file.
- **Task tracker CSV** — one row per Action (step / precondition / outcome / owner / due date / status / success criteria), for pasting into Jira / Trello / Planner / Asana. Only surfaces when the doc has Action entities — a Transition Tree is the canonical source.
- **Prerequisite plan CSV** — one row per Intermediate Objective in dependency order (objective / overcomes / depends-on / owner / due date / status / notes). Only surfaces when the doc has Intermediate Objectives — a Prerequisite Tree is the canonical source.
- **Risk register (CSV)** — one row per UDE in the doc, with columns `risk_id / risk / trigger / consequence / mitigation / owner / status`. The exporter walks each UDE backwards through the causal graph to find any reachable injection or desired-effect; if one exists the row is `mitigated`, otherwise `open`. Owner comes from `entity.owner` (with a back-compat fallback to `entity.attributes.owner.value` for older docs). Imports cleanly into Jira / Linear / a spreadsheet. Only surfaces in the Export picker when the doc has at least one UDE — NBR diagrams and CRTs are the canonical sources; an EC has no UDEs by construction so it's hidden there.
- **OPML** — outline form. OmniOutliner, Bike, Logseq.
- **DOT (Graphviz)** — for tooling pipelines.

- **Mermaid** — both export and import; round-trips with the Mermaid live editor.
- **VGL (declarative)** — Flying Logic-flavored declarative text.
- **Flying Logic XML** — round-trips with Flying Logic.

Annotations & reasoning

- **Reasoning narrative** — Markdown, sentence-per-edge prose. See Chapter 15.
- **Reasoning outline** — Markdown, nested-list form.
- **Annotations only (.md / .txt)** — just the description bodies, for content-review workflows.

Share

- **Copy read-only share link** — the whole document encoded into a URL fragment (covered below; also on the top-bar **Share** button).

Importing back

Going the other direction — opening someone else's file — uses the single **Import…** picker (`Cmd+K → Import…`). The dialog fans out five sources as cards: **TP Studio JSON** (full round-trip), **Flying Logic file** (`.logicx` / `.logic` / `.xlogic`), **Mermaid diagram** (`.mmd`), **Entities CSV** (append rather than replace), and **Paste from whiteboard** — the last is the escape hatch for Miro / Mural / FigJam / any text source. Copy stickies from the source board, paste into the textarea, one entity is minted per non-empty line. Bullet markers (`-` `*` `•` `1.` `1)`) are stripped, tab-separated content keeps only the first column. Connectors aren't inferred — Miro / Mural don't expose arrow structure in client-accessible exports, so this path gets the entities into the canvas; you wire causality after.

Save to file / Open from file

On Chromium browsers (Chrome / Edge), a trio of commands lets you work with a *real file on disk*. `Cmd+K → Open from file…` reads a `.tps.json` into a new tab. **Save to file** writes the current document back — and the first save (or an open) *remembers* the file, so every **Save to file** after that re-writes the same file **in one click, no dialog**. **Save to file as…** always opens the picker, for stashing a copy somewhere new. A small link-chip beside the document title shows which file you're bound to. It's all purely additive — the localStorage auto-save, the tabs, `Cmd/Ctrl+S` , and the Export/Import pickers behave exactly as before; this just adds a file on disk as a target for the same JSON.

> 💡 ***Put your trees on OneDrive.*** *Save into your synced* `OneDrive\…` *folder and the OneDrive client backs the file up and syncs it across your devices — no account linking, no setup.* ***Open from file…*** *the same file on another machine, edit, then* ***Save to file*** *to write straight back where you left off. (On Firefox / Safari, the commands are hidden — use* ***Export → JSON*** *and* ***Import…*** *for the same file, downloaded and uploaded.)*

Generating a diagram with an AI assistant

Sometimes the fastest first draft isn't drawing — it's *describing*. TP Studio ships a Claude **skill**, `tp-studio-import` , that turns a problem, goal, conflict or plan written in plain language into an importable TP Studio document. Tell Claude what you're looking at — "*a Current Reality Tree for why onboarding churns*", "*turn this dilemma into an Evaporating Cloud*" — and it emits a schema-valid `.json` for any of the nine diagram types, which you load through the same **Import… → TP Studio JSON** doorway above.

The skill lives in the repository at `.claude/skills/tp-studio-import/`, so it's available automatically to anyone using Claude Code inside the project; it can also be installed into Claude.ai or a personal Claude setup (its `SKILL.md` explains the format, `reference/format.md` the full schema). A guard test (`tests/skills/tpStudioImport.test.ts`) imports every bundled example through the real validator on each CI run — and fails if a new diagram or entity type is added without updating the skill — so the generated shape can't drift from what the app actually accepts.

Treat the output as a *first draft*. The assistant gets the entities and the causal skeleton onto the canvas in seconds; you still apply the CLR scrutiny of Chapter 13 that turns a plausible-looking tree into sound logic.

The U-Shape — linking trees into one journey

A folder of separate trees isn't a Thinking Process; the *journey* is. Cohen's spine is the **U-Shape**: pinpoint the **core problem** on a Current Reality Tree, surface the conflict underneath it as a **Core Cloud**, break the cloud with an **injection**, and check that injection's consequences in a **Future Reality Tree**. TP Studio stitches those separate documents into that one reasoning flow — without changing how any single tree is drawn.

Three opt-in moves, all on the palette (`Cmd+K`):

1. **Mark as core problem.** Select the CRT entity you've concluded is the core problem and mark it (also a one-click toggle in the inspector). This is *your* call — distinct from the app's computed "core driver" suggestion; it's the hinge the rest of the journey pivots on.
2. **Create the Core Cloud from this entity.** Opens a fresh Evaporating Cloud in a new tab, pre-tagged as a *Core cloud* and titled after the problem, already **linked back** to the CRT entity. Fill the five boxes, find the breakable assumption, draft the injection.
3. **Carry this into a new FRT.** From the injection, open a Future Reality Tree in a new tab with the injection seeded in, again linked back. Now grow the desired effects and trim any negative branch.

At every hop you get a new tab plus a reciprocal **"Linked to"** chip in the inspector. Click a chip to jump straight to the partner entity in its tab — so you can walk **CRT problem → Core Cloud → FRT injection** (and back) one click at a time, the U-Shape made navigable. The links are pure navigation: they add no causal arrows and change nothing about any individual diagram.

Share links

The top-bar **Share** button (or **Export… →** *Copy read-only share link*). Generates a URL that:

- Has the format `https://your-tp-studio-host/#!share=<base64-gzipped-json>`.
- Encodes the entire current document in the fragment.
- Never reaches a server — the fragment after `#` is client-side-only.
- On the receiver: TP Studio detects the `#!share=` fragment on boot, decodes, loads the doc, and **engages Browse Lock automatically** so the receiver can't accidentally commit edits to a foreign document.

Trade-offs:

- **Size**: a typical 20-entity CRT lands under 2 KB compressed; a 200-entity Goal Tree might push past 8 KB and trip length limits in some email clients. A toast warns when the link exceeds ~4 KB.
- **Visibility**: the fragment is in the receiver's browser history. Treat shared diagrams as "public enough to email" — same threat model as JSON export.
- **Hostility defense**: a 5 MB ceiling on the decompressed payload defends against gzip bombs.

Print preview

Export… → *Print / Save as PDF…*. Opens the **Print Preview Dialog** with:

- **Mode picker:** Standard (full color), Workshop (bold high-contrast), Ink-saving (no fills, lighter strokes). Each renders the same diagram differently for context.
- **Annotation appendix toggle:** include an addendum listing every entity description as a numbered footnote.
- **Header / footer:** plain text + `{title}` / `{pageNumber}` / `{pageCount}` merge fields.
- **Selection-only toggle:** print only the currently-selected entities (renders with non-selected nodes set to `visibility: hidden` so the canvas geometry stays intact).

Browser print dialog opens once you click Print.

The standalone HTML viewer

The most underrated export. **Export…** → *Self-contained HTML viewer* generates a single `.html` file that contains:

- The full TP Studio canvas runtime, bundled.
- The current document, base64-embedded in a `<script type="application/json">` tag.
- The current theme.

Double-clicking the file opens it in any browser. No network calls. No JavaScript dependencies. Read-only — the viewer has no edit gestures wired up.

Perfect for: airgapped audiences, security-conscious recipients, slides with embedded HTML, archival.

Reader mode: sharing a diagram with non-experts

Browse Lock keeps a reviewer from accidentally editing your diagram. Reader mode goes further: it reshapes the whole interface for someone who doesn't know TP notation at all.

Switch into it before handing a diagram to a manager, a client, or a domain expert who wasn't in the room when the tree was built: `Cmd+K → Switch to Reader mode`. Reader mode is a fifth app mode — alongside Expert, Guided, Workshop, and Presentation — that wraps a distraction-free, read-only shell on top of Browse Lock. The toolbar collapses to a single close button; the palette, inspector, and edit chrome all disappear. What remains is the diagram, a slim reading-hint banner across the top, and the canvas.

The banner shows the diagram-type reading rule in one sentence — for a CRT, that's something like *"Read bottom to top: lower nodes cause upper ones"*. For an EC it shows the five-box convention. The right framing in eight words is worth more than a four-paragraph email.

Coaching tooltips. Every entity and every edge gets a plain-language hover card. Hover a cloud box labelled "UDE" and the card says: *"Undesirable Effect — a symptom of the core problem."* Hover a causal arrow and it says: *"Cause–effect link: the lower node is claimed to produce the upper one."* The tooltips don't require any setup; they derive from element type and diagram context automatically.

"Challenge this arrow." Non-experts often sense that something is wrong before they can articulate it. Each edge in reader mode shows a small **Challenge this arrow** affordance — a flag icon on hover. Clicking it opens the comment composer in *reservation-first mode*: before the reviewer types a free-text note, the composer offers a short menu of reservation types drawn from the CLR taxonomy — *Causality existence*, *Cause sufficiency*, *Additional cause*, and so on — with a one-line plain-language gloss on each. They don't need to know the CLR; they just pick

the closest match. The reservation type pre-fills the comment, and the comment lands in the diagram's thread list exactly like any other review comment.

Worked example. You've built a CRT explaining why customer churn is rising. You want a sign-off from the VP of Sales, who has never seen a tree before. You enter Reader mode and paste the share link into an email. She opens it. The banner tells her to read from the bottom up. She hovers a node marked *"UDE"* — the tooltip tells her it's a symptom, not a root cause. She follows the arrows upward and hits a jump that bothers her: an arrow running from *"sales cycle lengthened"* straight to *"renewal pipeline dried up"* with nothing in between. She clicks Challenge this arrow, scans the reservation menu, picks *"Cause–effect existence — does this link really hold?",* and types *"We lengthened cycles on purpose to land bigger contracts. Is churn the actual outcome?"* The comment arrives in your Open list, pinned to that exact edge, already labelled with the CLR category she chose. You have a precise, actionable objection — not a vague "I'm not sure about this bit."

Exit Reader mode at any time with the **X** in the top-right corner; the diagram reopens exactly as you left it.

Review comments

Sharing a diagram for feedback used to mean the feedback came back *somewhere else* — a reply email, a Slack thread, a track-changes Word doc — detached from the diagram it was about. **Review comments** keep the feedback inside the file.

A comment is a short note pinned to one of three places: an **entity**, an **edge**, or the **whole diagram**. Open the panel with the speech-bubble button in the TopBar (or `Cmd+K → Comments`), select the thing you want to talk about, type, and post. With nothing selected the comment attaches to the diagram as a whole.

Because comments live in `doc.comments` — part of the document, not a side-channel — they travel with **every** lossless route out: JSON export, the read-only share link, and the standalone HTML viewer all carry the threads with them. The async-review loop becomes: share the link (or send the JSON) → the reviewer pins objections to the causality where it's wrong → they send it back → you open it and see each note attached to the exact entity or edge in question.

Each thread supports one level of **replies**, a **resolve / reopen** toggle, and inline **edit / delete**. The panel filters to **Open**, **Resolved**, or **All**, so a review pass is simply "work the Open list to zero." Click a thread's anchor chip to jump the canvas to whatever it's pinned to. Comments are signed with the name you set in the panel's *Signing as* field — a local label, not a login; TP Studio has no accounts.

Two boundaries worth knowing:

- **Comments are plain text.** No formatting, no `@`-mentions, no notifications — they're review notes, not a chat system.
- **Deleting the anchor deletes the comment.** Remove an entity or edge and any comments pinned to it go with it (whole-diagram comments stay). That keeps the thread list honest — every open comment points at something that still exists.

Sidebars

> ***🛠 How TP Studio helps***
>
> - `Cmd+K → Export` — *unified Export Picker.*
> - `Cmd+K → Copy read-only share link` — *fragment-encoded URL share.*
> - `Cmd+K → Print / Save as PDF` — *Print Preview Dialog.*
> - ***Standalone HTML viewer*** *— self-contained share artifact.*
> - ***EC Workshop Sheet PDF*** *— one-page handout for EC workshops.*
> - ***Browse Lock auto-engages on share-link load*** *— receivers can't accidentally edit.*
> - `Cmd+K → Switch to Reader mode` *— distraction-free shell for non-expert reviewers: reading-hint banner, coaching tooltips, and "Challenge this arrow" affordance.*
> - ***Review comments*** *— notes pinned to an entity / edge / the whole diagram, carried inside every JSON / share-link / HTML export.*
> - `tp-studio-import` ***Claude skill*** *— describe a problem in words; get an importable JSON for any of the nine diagram types.*

> ***💡 Practitioner tips***
>
> - ***Capture a snapshot before exporting for stakeholders.*** *"What I showed them on Tuesday" is a revision you'll want later.*
> - ***PDF for static audiences, share link for interactive ones.*** *Stakeholders click PDFs; analysts click links.*
> - ***Use redacted JSON when sharing the shape of an analysis.*** *"Here's our diagnostic shape, with anonymized content" is a real workflow.*
> - ***The EC Workshop Sheet is great handout material.*** *Print one per participant.*
> - ***Switch to Reader mode before sending to a domain expert.*** *They shouldn't need to learn TP notation to give you useful feedback.*
> - ***Use comments for async review.*** *Share the link, let reviewers pin objections to the exact edge, then work the Open filter to zero.*

> ***⚠ Common mistakes***
>
> - ***Sharing the wrong revision.*** *When you send a link, the link encodes the current doc state — not whichever snapshot you thought it would be. Capture + restore the right revision first.*
> - ***Forgetting Browse Lock when demoing locally.*** *Click the lock icon before screen-sharing. Otherwise an errant double-click during the demo creates a stray entity.*

🔁 **Chain to next:** sharing is the asynchronous mode. Workshops are the synchronous one. Next chapter covers the latter.

→ Continue to Chapter 17 — Workshops with TP Studio

Chapter 17 — Workshops with TP Studio

> *TOC workshops have a particular shape. Three to eight participants, one facilitator, four to six hours, one CRT or one EC built collaboratively. The patterns that make them work are mostly about discipline and rhythm; the tool plays a supporting role.*

Setting up the room (or the call)

Before the workshop opens, you'll want:

- **Browse Lock OFF.** You're going to edit live. Make sure the lock icon is unlocked. (Lock it during breaks if you're walking away from the laptop.)
- **Reading Instructions strip visible** on EC docs — `Cmd+K → Toggle EC reading guide` if it's currently hidden. New participants need the 1/2/3 reminder of EC reading direction.
- **System Scope filled.** Document Inspector → System Scope section. Answer the seven CRT-Step-1 questions before you start. "What system are we analyzing?" "Who are the stakeholders?" "What's the boundary?" The workshop will be 30% less productive if you skip this.
- **Method Checklist visible** in the Document Inspector. The canonical recipe is right there; refer back to it during the workshop to keep the group on the rails.

For remote workshops:

- **One person drives.** Screen-shared. Everyone else watches.
- **Use a video call with view-only screen share, not a tool that lets remote participants click on your canvas.** Multi-cursor editing of a CRT is chaos.
- **Capture snapshots at every transition** — between CRT and EC, before the negative-branch hunt, before re-reading. The snapshot history is the workshop's memory.

Facilitator gestures

Six gestures the facilitator uses repeatedly:

1. **Zoom-up annotation.** When a participant says "wait, that one — the third one from the left," hover over the node and let the zoom-up overlay surface its full title and description. Solves the "which entity?" problem instantly.
2. **Walkthrough overlay** (`Cmd+K → Start read-through`). When the group has been building for 20 minutes and is starting to lose the thread, switch to walkthrough. Reading the diagram aloud one edge at a time *re-centers* the group on what's been said.
3. **Side-by-side compare.** When a structural choice is being debated ("should we group these as AND or as two separate causes?"), branch first, try option A, snapshot, restore, try option B, snapshot. Now compare the two. The right answer is often obvious; the wrong answer was a function of having only seen one.
4. **CLR walkthrough.** Near the end of the workshop, run `Cmd+K → Start CLR walkthrough` to surface every open reservation. Address each as a group — even the dismissals are conversations.
5. **EC Workshop Sheet PDF.** For EC workshops, generate the workshop sheet PDF and print copies before the session. Each participant gets one; they write candidate assumptions on it during the cloud-construction phase, you transcribe the best onto the canvas.

6. **Park objections as comments.** When a participant raises a doubt the group isn't ready to resolve — "I'm not convinced that cause is *sufficient*" — drop a comment on the edge (`Cmd+K → Comments`, or the speech-bubble button) rather than stalling the build. The note pins to the exact edge and stays out of the way. At the closing CLR pass, open the Comments panel's **Open** filter and work through every parked note as a group — resolving each one is its own small conversation. Nothing real gets lost to the pace of the room.

A 4-hour CRT workshop — example agenda

This is one viable shape. Adapt to your context.

Time	Activity	TP Studio gesture
0:00–0:15	Intro + ground rules. Verbalisation discipline explained.	—
0:15–0:30	System Scope. Fill the seven questions live.	Document Inspector → System Scope
0:30–1:15	UDE brainstorm. Each participant offers 1-3 UDEs. Capture as `Undesirable Effect` entities.	Double-click + Type → UDE
1:15–1:30	Break + verbalise. Read aloud all UDEs in sequence.	—
1:30–2:30	First cause chain (highest-impact UDE). Ask why. Build downward. Capture snapshot when done.	`Cmd+K → Capture snapshot`
2:30–2:45	Break.	—
2:45–3:15	Second cause chain. Look for convergence with the first.	—
3:15–3:45	Find core driver. CLR walkthrough. Dismiss with notes.	`Cmd+K → Find core driver(s)`, then `Start CLR walkthrough`
3:45–4:00	Walk through final CRT. Capture final snapshot. Export reasoning narrative for distribution.	`Start read-through` then `Export → Reasoning narrative`

The EC workshop is shorter — typically 2 hours — but follows the same shape: scope, build, verbalise, validate.

What goes wrong, and what to do

- **One participant dominates.** The verbalisation discipline helps. So does rotating who reads aloud. If one voice is over-claiming, hand them the read-aloud job for the next five minutes.
- **The group disagrees on a cause.** Branch the doc and explore both. Reconverge at the snapshot.
- **A cause-claim "feels wrong" but no one can articulate why.** That's a CLR check waiting to be raised. Open the Inspector's Warnings list on the contested edge; usually a `cause-effect existence` or `cause-effect reversal` warning is already firing. If none fires, drop a comment on the edge so the doubt is captured and revisited at the closing pass rather than lost.
- **The workshop runs long.** Snapshot, set a hard stop, schedule a follow-up. A half-finished CRT is more valuable than a forced-completion one.

- **The CRT looks like a project plan.** You're confusing CRT (diagnosis) with PRT/TT (planning). Re-set: this diagram is *why things are the way they are now*, not *what we'll do about it*.

After the workshop — from diagram to commitment

The diagram is not the deliverable; the *change* is. A CRT everyone nodded at and nobody acted on was a nice afternoon, not an intervention. Three moves turn the room's work into commitment:

1. **Close on the next tree, not on applause.** A CRT workshop should end by naming the core problem and agreeing the next step — usually *Create the Core Cloud from this entity*, carrying the U-Shape forward (Chapter 16). Book the follow-up before everyone stands up; the analysis cools fast.
2. **Assign owners on the canvas.** Each injection or intermediate objective gets an **Owner** in the inspector and, where it's a plan, a due date. Export the **Task tracker CSV** (TT) or **Prerequisite plan CSV** (PRT) straight into the team's tracker so the diagram becomes the backlog, not a screenshot in a deck.
3. **Distribute the reasoning, not just the picture.** Send the **reasoning narrative** export + a PDF within 24 hours, and the **share link** to anyone who couldn't attend. Stakeholders who can re-read the logic — and **Challenge this arrow** in Reader mode — stay bought in; a picture with no argument attached is forgotten by Friday.

Then schedule the **re-measure**. The workshop named a gap (the Performance frame's Low / High); put a date on the calendar to come back and check it moved (Chapter 1 — Closing the loop). A TOC engagement that never re-measures can't tell a real win from a comfortable story.

Sidebars

> ***🛠 How TP Studio helps***
>
> - ***System Scope dialog*** *(Document Inspector) — "Step 0" for any analysis.*
> - ***Method Checklist*** *— the canonical recipe per diagram type, always visible.*
> - ***Zoom-up annotation*** *— readable titles at any zoom.*
> - ***App modes for the room*** *— `Cmd+K → Switch to Workshop mode` enlarges node text for a projector while you keep editing;* ***Presentation mode*** *gives a read-only, chrome-free projection with a bottom-centre* ***step-through*** *control (‹ / › or the arrow keys) for walking a finished tree past an audience without speaker notes.*
> - ***Walkthrough overlay*** *— re-centers the group on the diagram's reading.*
> - ***Side-by-side compare*** *— for structural disagreements.*
> - ***CLR walkthrough*** *— final discipline pass.*
> - ***EC Workshop Sheet*** *— printed participant artifact.*
> - ***Browse Lock*** *— read-only mode for demos and screen recording.*
> - ***Capture snapshot*** *— workshop memory.*
> - ***Review comments*** *— park objections on the exact entity/edge during the build; resolve them in the closing pass.*

> 💡 ***Practitioner tips***
>
> - ***Capture a snapshot every 20 minutes*** *in a workshop. Even unlabeled ones — you'll appreciate the rollback option.*
> - ***Read the diagram aloud at least three times*** *in a 4-hour workshop. Once after first cause chain, once after second, once at end.*
> - ***End with the exports.*** *Send the reasoning narrative + a PDF of the final CRT to participants within 24 hours.*
> - ***Collect async review with comments.*** *Send the JSON or share link to stakeholders who couldn't attend; their comments come back pinned inside the file, ready to work through.*

> ⚠ ***Common mistakes***
>
> - ***Skipping System Scope*** *because "we all know what we're analysing." You don't all know. The first 15 minutes are not optional.*
> - ***Multiple cursors on the canvas.*** *Cooperative editing of a TOC diagram does not work. One driver, one canvas.*
> - ***No verbalisation during build.*** *Just typing into entity titles is not a workshop; it's a brainstorm with a fancy outline tool. The aloud-reading is the gesture.*

🔁 **End of Part 4.** The appendices are reference material — keyboard shortcuts, CLR rule details, settings, a worked end-to-end case study, glossary, further reading.

→ Continue to Appendix A — End-to-end case study

Appendix A — End-to-end case study

Customer-support firefighting

> *The worked example used in Chapters 4–6 is sketched here as one continuous narrative. Read it once in full to see how the CRT, EC, and FRT compose into a single analysis.*

The scenario

A B2B SaaS company. The product is mature; the install base is ~400 customers paying an average of $40K ARR. Customer Success and Support are organizationally separate; Support has 8 agents (2 L1, 4 L2, 2 L3) and one team lead (formerly L3, promoted 14 months ago).

The VP of CS calls a meeting. Three concerns:

- Renewal pull-through is down. Net retention has dropped from 112% to 94% over the last 4 quarters.
- The Support team is burnt out. Three resignations in the last 6 months, all citing "constant firefighting."
- Cost per ticket — measured in agent-hours per resolution — is up 40% YoY despite no change in product complexity.

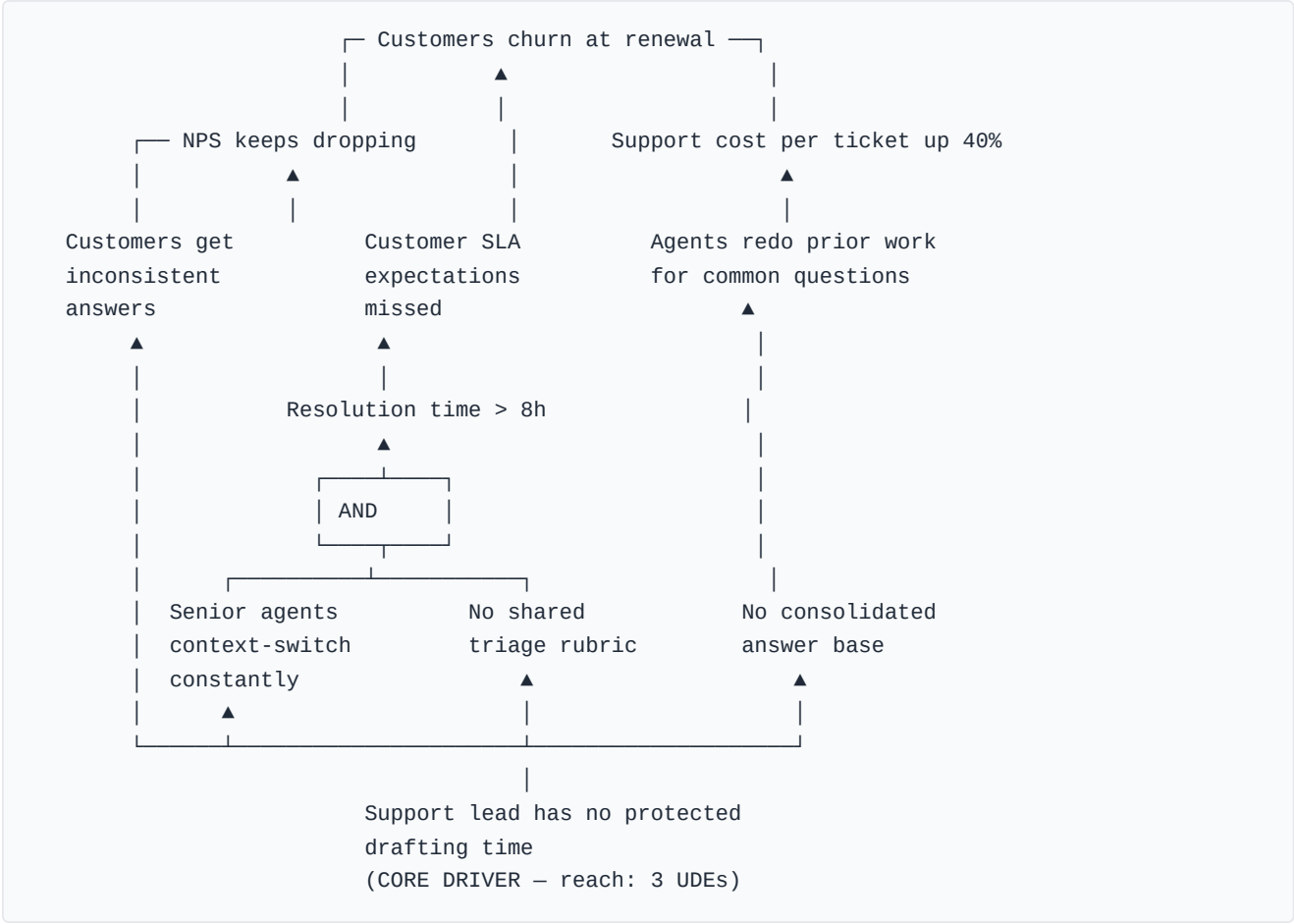
The conventional moves have already been tried. Hiring more agents didn't help. A new ticketing tool didn't help. A re-org didn't help. The VP wants a diagnosis.

CRT

Three UDEs:

- **Customers churn at renewal**
- **NPS keeps dropping**
- **Support cost per ticket up 40%**

Cause chains (built bottom-up):



The structure converges: three UDEs trace through five intermediate effects to two terminal causes ("Senior agents context-switch constantly" and "Support lead has no protected drafting time"), with the lead's protected-time problem feeding three of the four intermediate effects. The core driver is structural — not a hiring problem, not a tooling problem, not a re-org problem. The lead has not been protected from incoming work long enough to build the team's scaling apparatus.

The CLR check. Before trusting the diagnosis, the team scrutinised the load-bearing arrows — the discipline of Chapter 13. The edge *"Senior agents context-switch constantly"* → *"Resolution time > 8h"* drew a **cause-sufficiency** reservation: was context-switching *alone* enough? No — it AND-joins with *"No shared triage rubric"* (without a rubric, every interruption restarts from zero). They also ran **Scrutinize this edge** on the core driver's outgoing arrow and asked the reversal question — is it *"no protected time → no rubric"*, or *"no rubric → no protected time"*? — and confirmed the direction held. Two minutes of CLR turned a plausible tree into a defensible one.

EC

Why has this persisted? The EC.

- **A — Goal:** A sustainable support function.
- **B — Need:** Keep ticket queue responsive to retain at-risk customers.
- **C — Need:** Build durable structure (rubric + answer base) so the team scales.
- **D — Want (satisfies B):** Support lead stays on the queue.
- **D′ — Want (satisfies C):** Support lead takes 2 days/week off the queue to build structure.
- **Mutex:** D and D′ conflict — one person.

The cloud reads aloud:

> *In order to achieve a sustainable support function, we must keep ticket queue responsive to retain at-risk customers; therefore the support lead stays on the queue. In order to achieve a sustainable support function we must also build durable structure (rubric + answer base) so the team scales; therefore the support lead takes 2 days/week off the queue. And these two wants conflict.*

Assumptions on B → D:

- *Only the support lead can resolve the hard tickets.*
- *Tickets sufficient to keep the queue responsive must all be resolved by humans.*
- *We can't temporarily reduce inbound ticket volume.*
- *No structural improvement could pay off within the timeframe leadership cares about.*

Assumptions on C → D′:

- *Building the structure requires the lead specifically — no other agent can do it.*
- *The structure must be built in dedicated blocks, not incrementally.*

The breakable assumption is **"Only the support lead can resolve the hard tickets."** Trained L2 agents could handle the hardest 20% of ticket types, freeing the lead's time without sacrificing queue responsiveness.

The cloud evaporates. Injection: **Train 2 L2 agents on the hardest 20% of ticket types**.

FRT

Does the injection actually work?

Primary chain:

- Inject: Train 2 L2 agents → L2 agents handle hardest 20% → Lead's queue load drops 30% → Lead has 2 days/wk drafting time → Rubric + answer base exist → Resolution time drops, no answer-redoing → SLA met + cost per ticket drops → **CRT UDEs eliminated.**

Negative branches:

- **NB1:** L2 training takes time. Queue load on existing seniors spikes during training. → Inject: Temp contractor for 8 weeks during training.
- **NB2:** L2-resolved tickets might be lower quality than L3. → Inject: 2-week shadowing program before L2s go solo.

Three injections total, ready for the PRT.

PRT (sketch)

Below each injection, the obstacles + IOs. For the primary injection:

- *Obstacle:* No list of "hardest 20%". → *IO:* Audit complete; hardest 20% defined.
- *Obstacle:* L2 candidates have full queues. → *IO:* L2 capacity freed by re-routing.
- *Obstacle:* No curriculum. → *IO:* Curriculum drafted, lead-reviewed.
- *Obstacle:* No budget for contractor. → *IO:* Budget approved.

For the temp-contractor injection: → *IO:* Contractor identified, contracted, onboarded. (Standard recruiting / contracting workflow; mostly off-canvas.)

For the shadowing injection: → *IO:* Shadowing rubric drafted. *IO:* L3 calendars cleared 2 hours/day during shadowing weeks.

TT (sketch)

The most operationally explicit. One example chain from the PRT's "Audit complete" IO:

1. **Pull 6mo of escalated tickets via helpdesk API.** Precondition: API token. Outcome: CSV of ~2400 tickets.
2. **Bucket the CSV by type and compute escalation rate.** Precondition: CSV + categorization rubric. Outcome: Ranked list.
3. **Mark top 20%, circulate to L2 candidates.** Precondition: Ranked list + L2s identified. Outcome: Signed-off list.
4. **Publish list as workspace doc.** Precondition: Signed-off. Outcome: IO achieved.

Each leaf is sized for one person, one day at most.

The result

Six weeks after the workshop:

- L2 training in week 4, all 2 L2s certified on hardest-20% types.
- Lead has had 5 weeks of 2-days-protected time. Rubric drafted, sections 1-3 of answer base done.
- Resolution time has dropped from a median of 11 hours to 6.5.
- One renewal saved that the VP's gut said would have churned. Another is at risk but the underlying complaint shifted from "you're slow" to product-specific gaps — addressable in the next product cycle.

Twelve weeks later, both NB injections also paid out: the temp contractor extended for 4 weeks beyond plan because the lead wanted more drafting time; the L2 shadowing turned into a permanent pairing convention, with one L2 explicitly on a promotion track.

The CRT was a structural diagnosis. The EC named the false assumption. The FRT predicted the result. The PRT/TT planned the rollout. None of these was the answer alone; the chain was.

Appendix B — Keyboard reference

> *Mirrors the Help dialog (the `?` icon in the TopBar, or `Cmd+K → Help & keyboard shortcuts`). Reproduced here for offline reference.*

Canvas

Shortcut	Action
`Double-click` (empty canvas)	Create a new entity at click point
`Click` (entity / edge / group)	Select
`Shift+click`	Add to selection
`Cmd/Ctrl+click`	Toggle selection
`Alt+click` (on another entity, with one entity selected)	Create edge from selected → clicked
`Drag` (empty canvas)	Marquee-select
`Drag` (handle on entity edge)	Create edge to drop target
`Alt+drag` (entity onto edge)	Splice the dragged entity into the edge
`Middle-click drag` / two-finger scroll	Pan
`Wheel`	Zoom
`+` / `-` / `0`	Zoom in / out / fit-to-view
`Esc`	Cascade dismiss (close palette → close picker → exit search → clear selection)

Selection-driven

Shortcut	Action
`Tab` (entity selected)	Create child below the selection
`Shift+Tab` (entity selected)	Create parent above the selection
`Enter` / `F2` (entity selected)	Enter inline title edit
`Alt+Enter` (inside inline editor)	Newline in title
`Esc` (inside inline editor)	Cancel without committing the in-progress edit
`A` (edge selected)	Add an assumption to the edge
`Delete` / `Backspace`	Delete selection (confirms when there are connected edges)
`Cmd/Ctrl+C` / `X` / `V`	Copy / cut / paste
`Cmd/Ctrl+Shift+→`	Select all successors
`Cmd/Ctrl+Shift+←`	Select all predecessors
`Cmd/Ctrl+Shift+S`	Swap the two selected entities
`Cmd/Ctrl+D`	Duplicate the selection in place (doesn't touch the clipboard)
`↑` / `↓` / `←` / `→` (entity selected or focused)	Walk to the connected neighbour in that direction

On a selected group

Shortcut	Action
`Enter`	Hoist into the group
`→`	Expand (if collapsed)
`←`	Collapse (if expanded)
`Delete` / `Backspace`	Delete the group (members preserved)

Document-wide

Shortcut	Action
`Cmd/Ctrl+Z` / `Cmd/Ctrl+Shift+Z`	Undo / redo
`Cmd/Ctrl+S`	Save (force a flush + confirmation toast)
`Cmd/Ctrl+P`	Print / Save as PDF
`Cmd/Ctrl+K`	Open command palette
`Cmd/Ctrl+F`	Open find panel
`Cmd/Ctrl+\`	Close the inspector (clears the selection)
`Cmd/Ctrl+A`	Select every entity in the document
`Cmd/Ctrl+,`	Open settings
`Cmd/Ctrl+E`	Palette pre-filtered to Export commands
`E` (no modifiers, not in text field)	Open Quick Capture
`Cmd/Ctrl+T`	New tab *(installed app only)*
`Cmd/Ctrl+W`	Close tab *(installed app only)*
`Cmd/Ctrl+1` – `9`	Switch to tab 1–9, 9 = last *(installed app only)*

> *The three tab keys above fire only in an installed PWA (`display-mode: standalone`). In a normal browser tab those keys belong to the browser, so use the palette tab commands below instead.*

Palette commands worth memorising

Command	Purpose
`New diagram…`	Diagram type picker
`Load example…`	Example picker
`Browse templates…` / `New from template…`	The unified Templates library
`Capture snapshot`	Save a revision
`Comments`	Review-comments panel toggle
`Add comment on selection`	Comment on the selected entity / edge (or the whole diagram)
`Start CLR walkthrough`	Iterate open warnings
`Start read-through`	Verbalisation overlay
`Find core driver(s)`	Highest-reach root cause
`Spawn Evaporating Cloud from selected entity`	CRT → EC pivot
`Splice entity into selected edge`	Create-and-splice
`Group selected edges as AND` / `OR` / `XOR`	Junctor grouping
`Group selected entities`	Generic group
`Move selection to Archive group`	Quick archive
`Toggle EC reading guide`	EC-only
`Reopen creation wizard`	If you dismissed and want it back
`New tab` / `Duplicate tab` / `Close tab` / `Next tab` / `Previous tab`	Tab management (works in any browser)
`Forget closed documents`	Reclaim storage from documents you've closed
(Share button / `Export…` → Share)	Fragment-encoded share URL
`Export`	Open the unified picker

The complete list is in the palette itself — open `Cmd+K` and scroll. Categories are visible at the right edge of each row.

Appendix C — The CLR rules in detail

> *One entry per implemented validator. Each rule carries a **tier** (`clarity`, `existence`, or `sufficiency`) and a set of diagram types it fires on. Tier governs how the Inspector's Warnings list groups the rule — under **CLARITY**, **EXISTENCE**, or **SUFFICIENCY** headers — matching how TOC practitioners talk through reservations in a workshop.*

TP Studio implements **three tiers**, not the classical eight CLR categories one-for-one. The categories are a *teaching* taxonomy (see Chapter 13); the tiers are how the tool *groups what it can detect automatically*. Many rules map onto a classical category (`cause-effect-reversal` ↔ Cause-effect reversal); others are structural build-quality checks the classical list never named (`crt-dead-branch`, `long-arrow`). A rule fires only when a pure predicate can detect its trigger — most of the CLR is too contextual for that, which is why **edge scrutiny** exists to walk every category by hand.

Structural rules (every diagram type)

These read titles, edge endpoints, and connectivity only — they assume nothing about which entity types exist, so they run on all nine diagram types (CRT, FRT, PRT, TT, EC, Goal Tree, S&T, Freeform, NBR).

Rule	Tier	Fires on	Catches
`clarity`	clarity	Any non-note entity	A title over 25 words (tighten to one statement) **or** a title ending in `?` (make it declarative).
`entity-existence`	existence	Any entity	An empty title — the slot asserts a state of the world but doesn't say what.
`causality-existence`	existence	Each edge	A standing once-per-edge reservation: does this drawn arrow correspond to something real? Resolve it once you're confident in the link.
`tautology`	clarity	Each edge	A cause that merely restates its effect — a relabel, not a causal step.
`indirect-effect`	existence	Converging edges	Too many causes pointing straight at one effect — the *breadth* twin of `long-arrow`; a consolidating intermediate effect is probably missing.

> *There is no `cycle` rule: loops are auto-detected and the loop-closing edge renders as a back-edge, so a separate warning would be redundant. The loop-focused lint below (`loop-polarity`, `reinforcing-no-delay`) interrogates a loop's dynamics instead.*

Diagram-specific rules

CRT — Current Reality Tree

Rule	Tier	Catches
`cause-sufficiency`	sufficiency	A sufficiency edge that probably needs a co-cause — where AND-groups are born.
`additional-cause`	sufficiency	A UDE that a *different*, independent cause could also produce (model with an OR-junctor).
`cause-effect-reversal`	existence	An edge whose typed reading suggests the arrow points the wrong way.
`external-root-cause`	clarity	A root cause flagged Locus = **External** — push one level deeper; the real driver is usually within control or influence.
`crt-ude-count`	clarity	A CRT scoped to too few (< 3) or too many (> 15) UDEs.
`crt-ude-no-upstream`	existence	A UDE with no incoming cause — the tree is incomplete there.
`crt-dead-branch`	clarity	A non-UDE entity that leads to no UDE — trim it, or connect it into the chain.
`crt-low-core-driver-coverage`	clarity	The leading root cause explains fewer than half the UDEs — the tree may hold two independent clusters.
`crt-tied-core-drivers`	clarity	Two or more root causes tie for the most UDEs — a hidden conflict may sit beneath. Carries a one-click **Spawn Evaporating Cloud** action.
`crt-ude-wording`	clarity	A UDE phrased as the *absence of a solution* ("lack of…", a leading "No…") rather than an observable effect.

FRT — Future Reality Tree

Rule	Tier	Catches
`cause-sufficiency`	sufficiency	An injection/cause that probably needs a co-cause.
`additional-cause`	sufficiency	A desired effect a different cause could also produce.
`predicted-effect-existence`	existence	An injection whose predicted *second* effect should be observable somewhere — confirm it exists, or the claim is suspect.

NBR — Negative Branch Reservation

Runs the FRT-style set (a negative branch is an FRT subtree that ends in UDEs): `cause-sufficiency`, `additional-cause`, and `predicted-effect-existence` (all as above), plus two NBR-specific shape rules and the cross-diagram lint below.

Rule	Tier	Catches
`nbr-no-negative-branch`	existence	No undesirable effect captured yet — trace the injection forward to where the chain turns negative ("yes, but…"). Without a UDE this still reads as an FRT. Document-level.
`nbr-ude-disconnected`	existence	A UDE that doesn't trace back to the candidate injection — connect the chain (injection → … → UDE) or it can't inform the adopt / modify / reject call.

PRT — Prerequisite Tree

Rule	Tier	Catches
`prt-obstacle-no-io`	existence	An obstacle with no Intermediate Objective overcoming it — add the IO that removes it on the way to the goal.
`prt-io-no-obstacle`	existence	An Intermediate Objective that doesn't overcome any obstacle — connect it to the obstacle it removes.

TT — Transition Tree

Rule	Tier	Catches
`complete-step`	sufficiency	An Action whose edge to its Outcome has no precondition feeding it — what existing condition lets the action produce the outcome?
`tt-action-locus-unset`	clarity	An Action with no **Locus** set (control / influence / external) — state it in a way you can act on.

EC — Evaporating Cloud

Rule	Tier	Catches
`ec-missing-conflict`	existence	Neither D ↔ D′ edge carries the lightning-bolt mutex marker — the conflict isn't declared.
`ec-completeness`	existence	The brief's structural set: an empty Objective (A); Needs B and C collapsing into one entity; a Need connected to anything other than A; a Want supporting the wrong Need (D → B only, D′ → C only); an arrow with no assumption recorded; no injection captured yet.

S&T — Strategy & Tactics Tree

Rule	Tier	Catches
`st-tactic-assumptions`	clarity	A tactic with fewer than three Necessary-Condition feeders (the NA / PA / SA facet pattern).
`st-tactic-rollup`	sufficiency	A non-apex tactic with no child tactics — a layer that should decompose but doesn't.

Goal Tree

Rule	Tier	Catches
`goalTree-multiple-goals`	clarity	More than one apex Goal entity. Soft + dismissible; carries a one-click **Convert extras to CSFs** action.
`goalTree-csf-no-ncs`	sufficiency	A Critical Success Factor with no Necessary Conditions beneath it — add the conditions that must hold for it.
`goalTree-csf-count`	clarity	A CSF count outside Dettmer's typical 3–5 band (document-level; silent at zero). Too few suggests missing make-or-break conditions; more than 5 usually means some are really NCs a tier down.

Cross-diagram lint (the System-Dynamics lens)

These ride the same edge/loop structure across several diagram types:

Rule	Tier	Fires on	Catches
`logic-type-mismatch`	clarity	CRT · FRT · TT · EC · Goal Tree · NBR	An edge whose kind (sufficiency vs necessity) contradicts the diagram's primary logic.
`loop-polarity`	clarity	CRT · FRT · NBR	A balancing (self-correcting) loop where a reinforcing (self-amplifying) one is expected, or vice versa.
`long-arrow`	existence	CRT · FRT · TT · NBR	A sufficiency edge skipping three or more causal levels — the *depth* twin of `indirect-effect`. Carries a one-click **Insert a step** action.
`reinforcing-no-delay`	clarity	CRT · FRT · NBR	A reinforcing loop none of whose edges carries a delay marker — it would escalate instantly; a lag is probably un-modelled.

Diagram-type scoping matrix

`✓` = the rule runs on that diagram type. Blank = it doesn't. Freeform runs the structural rules only.

Rule	CRT	FRT	PRT	TT	EC	Goal	S&T	Free	NBR
`clarity`	✓	✓	✓	✓	✓	✓	✓	✓	✓
`entity-existence`	✓	✓	✓	✓	✓	✓	✓	✓	✓
`causality-existence`	✓	✓	✓	✓	✓	✓	✓	✓	✓
`tautology`	✓	✓	✓	✓	✓	✓	✓	✓	✓
`indirect-effect`	✓	✓	✓	✓	✓	✓	✓	✓	✓
`cause-sufficiency`	✓	✓							✓
`additional-cause`	✓	✓							✓
`cause-effect-reversal`	✓								
`predicted-effect-existence`		✓							✓
`external-root-cause`	✓								
`crt-ude-count`	✓								
`crt-ude-no-upstream`	✓								
`crt-dead-branch`	✓								
`crt-low-core-driver-coverage`	✓								
`crt-tied-core-drivers`	✓								
`crt-ude-wording`	✓								
`prt-obstacle-no-io`			✓						
`prt-io-no-obstacle`			✓						
`complete-step`				✓					
`tt-action-locus-unset`				✓					
`ec-missing-conflict`					✓				
`ec-completeness`					✓				
`st-tactic-assumptions`							✓		
`st-tactic-rollup`							✓		
`goalTree-multiple-goals`						✓			
`goalTree-csf-no-ncs`						✓			
`goalTree-csf-count`						✓			
`nbr-no-negative-branch`									✓
`nbr-ude-disconnected`									✓
`logic-type-mismatch`	✓	✓		✓	✓	✓			✓
`loop-polarity`	✓	✓							✓

Rule	CRT	FRT	PRT	TT	EC	Goal	S&T	Free	NBR
`long-arrow`	✓	✓		✓					✓
`reinforcing-no-delay`	✓	✓							✓

Freeform receives only the structural rules by design — see Chapter 11.

Appendix D — Settings reference

> *Everything in the Settings dialog (`Cmd/Ctrl+,` , or `Cmd+K → Settings…`), what it does, when to flip it. Four tabs: Appearance, Behavior, Display, Layout — the first three are app-wide preferences; Layout is per-document.*

Appearance

Setting	Default	What it does	When to change
Theme	Light	Light, Dark, High contrast, plus four dark variants: Rust / Coal / Navy / Ayu	Dark for evening work; High contrast for accessibility / projector use.
Color palette	Default	Default / Colorblind-safe (Wong palette) / Monochrome — recolours node stripes, edges, the minimap, and the building-blocks rail together	Colorblind-safe in mixed-audience workshops. Monochrome for print.

Behavior

Setting	Default	What it does
Animation speed	Normal	Instant / Slow / Normal / Fast. Normal follows the OS "reduce motion" accessibility hint; Instant skips fades entirely (good for screen recordings).
Browse Lock	Off	The read-only mode — same toggle as the ⋮ overflow's *Lock for browsing*.
Auto-snapshot while editing	On	Periodic `Auto` revision snapshots while you edit (only when the tree actually changed). See Chapter 14.
Creation wizards — Goal Tree	On	Auto-open the wizard on new Goal Tree docs.
Creation wizards — Evaporating Cloud	On	Auto-open the wizard on new EC docs.
Creation wizards — Current Reality Tree	On	Auto-open the wizard (capture your first three UDEs) on new CRT docs.
Selection toolbar	On	The floating 3–5 verbs above selected entities.
Open documents in new tabs	On	Importing, or loading a template / example / shared link, opens a new tab instead of replacing the current document. Off restores the pre-tabs "replace what's open" behaviour.

Display

Setting	Default	What it does
Show annotation numbers	Off	Renders each entity's `annotationNumber` as a small badge.
Show entity IDs	Off	Renders the entity's stable id as a tiny corner badge. Mostly for debugging.
Grow cards to fit text	Off	Lets an entity card grow taller (up to six lines) to show its full title instead of clamping to two.
Show UDE-reach badge	Off	Toggles the amber `→N UDEs` pill on each entity.
Show root-cause-reach badge	Off	Toggles the sky `←N root causes` pill.
Show action-eligibility badge	Off	Toggles the ✓ / ✗ / … eligibility pill on Transition-Tree Action nodes. Off by default — fresh TTs read "pending" everywhere until states are set.
Show minimap	On	Toggles the minimap with viewport indicator (node thumbnails tinted by entity type).
Ink-saving print mode	Off	Removes group fills and lightens strokes for cheaper printing.
Causality reading	Auto	None / Auto / Because / Therefore / In order to — fallback edge label when no per-edge label is set. See Chapter 3.
Default direction for new documents	Auto	Auto / Bottom→Top / Top→Bottom / Left→Right / Right→Left — the starting orientation for newly-created docs. Existing docs keep their own per-doc direction (change that on the Layout tab). EC ignores this (manual layout).
Layout density	Balanced	Compact / Balanced / Spacious — how tightly auto-layout packs the tree.
Edge routing	Smart	Smart (route around obstacles) / Direct (plain bezier curves through anything).

Layout (per-document)

Unlike the other tabs, these settings live **on the current document**, and only apply to auto-laid-out diagrams (CRT / FRT / PRT / TT / Goal Tree / S&T / NBR). Hand-positioned diagrams (EC, freeform) show an explanatory note instead.

Setting	Default	What it does
Direction	(per doc)	Bottom→Top / Top→Bottom / Left→Right / Right→Left — this is where you change an *existing* diagram's orientation.
Compactness	50	0–100 slider — denser packing vs. more breathing room.
Bias	Auto	Auto, or gravitate the tree toward one corner: Upper-left / Upper-right / Lower-left / Lower-right.
Reset to defaults	—	Appears when any of the above is overridden; clears the per-doc overrides.

The footer's **Restore defaults** button (confirm-gated) resets every app-wide preference at once.

Appendix E — Glossary

> *Terms used throughout. The TOC tradition is acronym-heavy; this list disambiguates.*

Term	Definition
AND junctor	A combinatorial node in TP Studio rendering "all inbound causes jointly sufficient." Visual: violet circle labeled `AND`.
Assumption	A claim that a causal arrow depends on, and that someone could plausibly challenge. An *edge annotation* (not an entity type) — added from the Edge Inspector's Assumption Well or by pressing `A` on a selected edge; renders as a violet card tied to its arrow.
Back-edge	The edge that closes a causal loop. Auto-detected (the loop-closer renders dashed with a `↻` glyph and an R/B polarity badge); can also be tagged manually to pick the closing edge or to name the loop. There is no cycle warning — the rendering *is* the acknowledgement.
Browse Lock	TP Studio's read-only mode. Toggle via the ⋮ overflow menu (*Lock for browsing*) or Settings → Behavior. Auto-engages on share-link load.
CLR	Categories of Legitimate Reservation. The discipline-checks for evaluating a causal claim — eight in Dettmer's teaching layout (Chapter 13).
Core driver	The root cause with the highest UDE-reach in a CRT — the candidate constraint.
CRT	Current Reality Tree. "Why is this happening?"
CSF	Critical Success Factor. Middle layer of a Goal Tree.
D / D′	The two "wants" in an Evaporating Cloud — the actions one side and the other side advocate.
DAG	Directed Acyclic Graph. Most TP Studio diagrams are DAGs (back-edges are explicit annotations of cycles, not structural cycles).
DE	Desired Effect. The FRT's top-of-tree entity (what the system would produce after the injection).
EC	Evaporating Cloud. The 5-box conflict diagram.
Effect	An entity in a CRT/FRT that's caused by something and causes something else — intermediate.
Evidence	A first-class list on every entity: structured rows backing the entity with a `description`, a 5-way `source` (`Observed / Stakeholder / Metric / Policy / Assumption`), a 3-way `strength` (`Weak / Moderate / Strong`), an optional URL, and a per-row validation stamp. Surfaces in the Inspector beneath the Owner block and feeds the `evidence` column of the Risk Register (CSV) export.
FRT	Future Reality Tree. "What would it look like solved?"
Goal	The top of a Goal Tree. Single (typically). Time-bounded (preferably).
Goal Tree	Top-down decomposition: Goal → CSFs → NCs. Strategic-planning shape.
Injection	A proposed change to the system. The hypothesis to test.
IO	Intermediate Objective. A state that, achieved, dissolves an obstacle. PRT-specific.
NA	Necessary Assumption. The first facet of an S&T card — "why this matters now."
NBR	Negative Branch Reservation. A forward-causal sub-tree from a candidate injection that maps its unintended consequences and the mitigation that breaks the chain. Available in TP Studio as both (a) a "Negative Branch" group preset inside an FRT for sub-branch capture, and (b) its own first-class **NBR** diagram type — `Cmd+K → New diagram… → NBR`.
NC	Necessary Condition. Lower layer of a Goal Tree.

Term	Definition
NPS	Net Promoter Score. Used in the case study as a UDE signal; not a TOC term.
OR junctor	"Any one of the inbound causes is sufficient." Visual: indigo circle.
PA	Parallel Assumption. The third facet of an S&T card — "why this specific approach."
Owner	Per-entity free-form text field naming whoever's accountable for the entity (decision owner, action assignee, validation owner). Feeds the `owner` column of the Risk Register (CSV) export.
Templates library	Curated starter diagrams for common TOC scenarios (~69, all diagram types). `Cmd+K → Browse templates…` (or Start page → Templates) lists them with a filter chip row; "+ Insert here" merges a same-type template into the current diagram. Distinct from "Load example…" which loads one canonical example per diagram type.
PRT	Prerequisite Tree. "What's in our way?"
Risk register	A tabular accounting of identified risks, one per row, with `risk / trigger / consequence / mitigation / evidence / owner / status` columns. TP Studio's **Risk register (CSV)** export (Chapter 16) generates one from any doc containing UDEs by walking each UDE backward through the causal graph to find reachable injections (the mitigations). The `evidence` cell renders the UDE's `evidence[]` entries as semicolon-joined `[strength/source] description (url)` rows. Status is `mitigated` if any mitigation reaches the UDE, `open` otherwise.
Root cause	A terminal cause at the bottom of a CRT — the leverage point.
S&T	Strategy & Tactics Tree. Operational-deployment decomposition with 5-facet cards.
SA	Sufficiency Assumption. The fifth facet of an S&T card — "why this tactic is enough."
Locus	Per-entity flag: `control` / `influence` / `external`. Previously labelled "Span of control" in TP Studio; the schema field name `spanOfControl` is retained for backward compatibility.
Start page	The workspace TP Studio opens on: a problem-led hero, the All-trees library (every saved tree, open or closed), Templates, a Needs-review triage view, and Learn-the-method links. The top-left logo returns to it from the editor.
Strategy	The second facet of an S&T card — the outcome-shaped "what."
Sufficiency edge	An edge claiming "this cause, by itself, produces the effect." Default for CRT/FRT/TT edges.
Necessity edge	An edge claiming "the effect requires this cause." Default for PRT/EC edges.
Tactic	The fourth facet of an S&T card — the concrete actions.
TT	Transition Tree. Action / precondition / outcome triples. "How do we get there?"
TOC	Theory of Constraints. Goldratt's framework.
UDE	Undesirable Effect. The symptom layer at the top of a CRT — what stakeholders / customers / the market actually feel.
VerbalisationStrip	The above-canvas paragraph rendering of an EC, updates live.
XOR junctor	"Exactly one of the inbound causes occurs." Visual: rose circle.

Appendix F — Further reading

> *The TOC literature. Goldratt's novels first; then the more rigorous works.*

Foundational — Goldratt's novels

These are the canonical entry points. Read in order of publication, not in order of process depth.

- **Eliyahu M. Goldratt, *The Goal: A Process of Ongoing Improvement* (1984).** The original. Manufacturing-focused but the mental model transfers everywhere. The Five Focusing Steps emerge here.
- **Eliyahu M. Goldratt, *It's Not Luck* (1994).** Introduces the Thinking Processes in narrative form — CRT, EC, FRT, PRT, TT. The chapter where the protagonist draws his first Evaporating Cloud is the best onboarding to ECs in any TOC literature.
- **Eliyahu M. Goldratt, *Necessary But Not Sufficient* (2000).** Applies TOC to enterprise software (ERP). Useful for software people.
- **Eliyahu M. Goldratt, *Critical Chain* (1997).** TOC applied to project management. The pattern transfers; the application is dated. Read for the patterns, skim the project-management specifics.
- **Eliyahu M. Goldratt, *The Choice* (2008).** Late-career meta-reflection. Read after the others.

Rigorous / academic

- **James F. Cox III & John G. Schleier Jr. (eds.), *The Theory of Constraints Handbook* (2010).** McGraw-Hill, ~1200 pages. The canonical reference. Each chapter is a topic by a different practitioner. The Thinking Processes chapters by William Dettmer are the deepest single source available.
- **William H. Dettmer, *The Logical Thinking Process: A Systems Approach to Complex Problem Solving* (2007).** Dettmer's expansion of Goldratt's TPs with more rigor and notation precision. The CLR is covered in clinical detail.
- **William H. Dettmer, *Goldratt's Theory of Constraints: A Systems Approach to Continuous Improvement* (1997).** The earlier, leaner Dettmer book. Often a better starting point than the 2007 expansion.
- **H. William Dettmer, *Strategic Navigation: A Systems Approach to Business Strategy* (2003).** Goal Tree and Strategy & Tactics applied to corporate strategy.
- **William H. Dettmer, *Thinking with Flying Logic* (2011).** A practitioner companion specifically for the Flying Logic software. The closest structural parallel to the present book — same shape (a practitioner-facing companion for a TOC software canvas), different canvas. Worth a read if you also use Flying Logic, or if you want to see how the same method-first / tool-second framing reads from inside the other tool's vocabulary.

TOC software lineage

- **Vector Goldratt site.** https://www.goldrattresearchlabs.com/ AGI's research arm. Papers, white papers, training material.

Practitioner blogs and field literature

- **Allan Dabrowski's writing on TOC implementations** — search for his Lean Six Sigma + TOC integration material.

- **Lisa Scheinkopf, *Thinking for a Change* (1999).** A practical workbook for using the TPs. Less novelistic than Goldratt, more accessible than Dettmer.
- **Eli Schragenheim, *Management Dilemmas: The Theory of Constraints Approach to Problem Identification and Solutions* (1998).** Worked TOC analyses across multiple industries; pattern-recognition material.

On the verbalisation discipline

There's no single source. The discipline is implicit in Goldratt's writing (he wrote his TPs as conversations because that's how TPs *work*), explicit in Dettmer's CLR chapters, and varies in name across practitioners ("read-aloud," "scrutinise," "challenge by reading"). The point survives all the terms.

Beyond TOC

Not strictly TOC, but adjacent and useful:

- **Peter Senge, *The Fifth Discipline* (1990; rev. 2006).** Systems Thinking — different vocabulary, similar instincts. The reinforcing-loop / balancing-loop notation is what TP Studio's back-edge feature owes to.
- **Donella Meadows, *Thinking in Systems* (2008).** The plainest introduction to systems thinking. Reads in an evening.
- **Russell L. Ackoff, *Re-Creating the Corporation* (1999).** Pre-TOC systems thinking applied to organizational design. Same instincts, different generation.
- **Edward Tufte, *Visual Display of Quantitative Information* (1983).** Tangentially relevant — the discipline of *what makes a graphic legible* applies to TOC diagrams as much as to any other.

A short reading order

If you want a guided sequence:

1. *The Goal* (Goldratt).
2. *It's Not Luck* (Goldratt).
3. This book.
4. *The Logical Thinking Process* (Dettmer).
5. Selected chapters from *The Theory of Constraints Handbook*.

Three weeks of evening reading, give or take. By the end you have a working TOC mental model that holds up in workshops.

Appendix G — Troubleshooting your diagram

> *A reverse index: start from the symptom — "this diagram feels wrong" — and work back to the cause and the fix. Where a TP Studio CLR validator catches the smell automatically, the rule is named (see Appendix C for the full registry). Many smells are too contextual for a rule, so the fix is a habit, not a warning.*

A finished diagram that's subtly wrong is more dangerous than an obviously unfinished one — it carries false confidence. This appendix collects the smells that recur in real work, what each usually means, and how to clear it.

Smells that apply to any causal tree

The smell	What it usually means	Catches it	The fix
It reads like a to-do list — every node is an action ("hire a PM", "buy the tool").	You've drawn a *plan* (PRT/TT), not a *diagnosis* (CRT).	— (judgment)	Restate each node as an *effect* or state of the world, not a deed: "Onboarding is slow", not "speed up onboarding".
A node could be glued to any tree — "Communication is poor", "There's no alignment".	The entity is too abstract to observe or challenge.	`clarity` (overlong titles only — vagueness slips under it)	Make it observable: what would you *see* if it were true? "Release notes ship after the release, not before."
An arrow makes you wince but you can't say why.	The link is doing too much work, or runs the wrong way.	`cause-effect-reversal`, `long-arrow`	Run **Scrutinize this edge** (Chapter 13) and ask each CLR question of it by hand.
One leap covers three steps — "We cut QA → customers churn."	A *long arrow* hiding unstated intermediate effects.	`long-arrow` (existence)	Use the one-click **Insert a step** action; name the missing middle.
A cause feels necessary but not enough.	A co-cause is missing.	`cause-sufficiency` (CRT/FRT/NBR)	AND-group the co-cause in — or accept the warning if it genuinely is sufficient.
The tree is a pile of disconnected fragments.	Separate analyses, or missing links.	`indirect-effect`	Connect them, or split into separate documents.
You dismissed a warning and can't recall why.	An undocumented judgment call — a debt for the next reader.	—	Dismiss *with* an explanation in the entity's description.

Per-diagram smells

Current Reality Tree

The smell	Catches it	The fix
Two or more root causes tie for "explains the most UDEs".	`crt-tied-core-drivers`	A hidden conflict may sit beneath — use the one-click **Spawn Evaporating Cloud** and dissolve it.
The leading root cause explains fewer than half the UDEs.	`crt-low-core-driver-coverage`	The tree may hold two independent clusters; split it, or keep digging for the deeper shared cause.
A UDE has no cause feeding it.	`crt-ude-no-upstream`	The tree is incomplete there — ask "why?" once more.
A UDE is phrased as a missing solution — "No triage rubric."	`crt-ude-wording`	Restate it as the *effect* of the absence: "Tickets are re-triaged from scratch each time."
Fewer than 3, or more than ~15, UDEs.	`crt-ude-count`	Too few and the analysis is thin; too many and you're boiling the ocean — cluster or scope down.
A leaf "root cause" flagged External.	`external-root-cause`	Push one level deeper; the real lever is usually inside your control or influence.

Evaporating Cloud

The smell	Catches it	The fix
The two Wants don't actually conflict.	`ec-missing-conflict`	If both can be true at once it isn't a cloud — you've drawn two parallel needs. Find the real either/or and mark the D ↔ D′ mutex.
The cloud is structurally incomplete (a missing need, edge, or assumption).	`ec-completeness`	Fill the five boxes, the four necessity edges, and an assumption per arrow.
No injection after the analysis.	—	A cloud you can't break is a complaint. Keep cycling assumption statuses until one reads **Invalid** — that's the lever.

Future Reality Tree / Negative Branch

The smell	Catches it	The fix
The FRT has no negative branches.	—	You haven't looked hard enough. For each injection, ask "what could this break?" and start a negative branch.
An injection's predicted second effect appears nowhere.	`predicted-effect-existence`	If A→B is real, B's other consequences should show too. Draw them, or doubt the claim.
An NBR has a forward trace but no UDE.	`nbr-no-negative-branch`	It still reads as an FRT. Follow the chain to where it turns negative ("yes, but…") and name the UDE.
An NBR UDE doesn't trace back to the injection.	`nbr-ude-disconnected`	Connect the chain (injection → … → UDE) — an off-chain UDE can't inform the adopt/modify/reject call.

Prerequisite Tree

The smell	Catches it	The fix
An obstacle has no Intermediate Objective overcoming it.	`prt-obstacle-no-io`	An obstacle without an IO is a complaint — write the state that dissolves it.
An IO doesn't overcome any obstacle.	`prt-io-no-obstacle`	Connect it to the obstacle it removes, or ask whether it's gold-plating.

Transition Tree

The smell	Catches it	The fix
An action has no precondition feeding its outcome.	`complete-step`	Name the existing condition the action relies on, or AND-group the co-cause.
An action's locus isn't set.	`tt-action-locus-unset`	Mark it control / influence / external — a step outside your control needs a different plan.

Goal Tree / Strategy & Tactics

The smell	Catches it	The fix
More than one apex Goal.	`goalTree-multiple-goals`	One-click **Convert extras to CSFs**, or genuinely split the analysis.
A CSF has no Necessary Conditions beneath it.	`goalTree-csf-no-ncs`	Add the conditions that must hold for it — a bare CSF is an assertion, not a plan.
Fewer than 3, or more than 5, CSFs.	`goalTree-csf-count`	Dettmer's band is 3–5: too few misses make-or-break conditions; too many usually means some are really NCs a tier down.
An S&T tactic has fewer than three assumption facets.	`st-tactic-assumptions`	Add the Necessary / Parallel / Sufficiency assumptions — the facets *are* the argument.
A non-apex tactic has no children.	`st-tactic-rollup`	Decompose it into the next level, or mark it a genuine leaf.

Feedback loops (any diagram with a cycle)

The smell	Catches it	The fix
An edge's kind fights the diagram's logic (a necessity arrow in a sufficiency tree, or vice versa).	`logic-type-mismatch`	Re-read the arrow aloud in both wordings; set the kind that matches the claim.
A loop reads as self-correcting where you expected a spiral (or vice versa).	`loop-polarity`	Check each edge's polarity around the loop — the R/B badge is the product of them all; one mis-signed edge flips the reading.
A reinforcing loop has no time lag anywhere.	`reinforcing-no-delay`	Real feedback lags. Mark the slow edge as delayed (`//`), or dismiss if the instant reading is genuinely intended.

The meta-fix: read it aloud

Most of these smells surface the instant you *verbalise* the diagram — `Cmd+K → Start read-through` (Chapter 15). The sentence that makes you hesitate is the smell; the rule, where one exists, just tells you which kind. When no rule fires and the sentence still reads wrong, trust the sentence.