A VISUAL REFERENCE · Graphviz

Graphviz, one change at a time

From a first edge to process maps, architecture, networks, data structures and deliberate diagram design. Complete source. A small change. Its actual result.

The Org file is the canonical source. Read the HTML guide beside your work, or download its PDF and source for offline reading.

> **Last changes: 2026-10-14.** Visual overview: browse all 108 diagrams across 20 chapters before the reading notes. Added 36 examples of architecture, trees, protocols, planning, graph algorithms and data structures. Every thumbnail opens a complete example with rendered SVG and downloadable DOT source.

108
worked examples

20
topic families

1
canonical Org source

All diagrams at a glance

Choose a diagram to open its complete DOT source, explanation and downloads.

First graphs

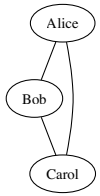
3 diagrams

Start with directed and undirected graphs, then recognize branches and joins.



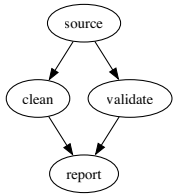
GVZ-01-001

Connect three steps



GVZ-01-002

Draw an undirected network



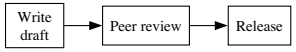
GVZ-01-003

Branch and join

Labels and styling

3 diagrams

Give nodes meaningful labels, set reusable defaults and explain decision branches.



GVZ-02-001

Separate names from labels



GVZ-02-002

Set defaults and override a node



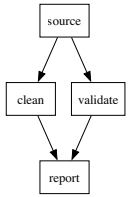
GVZ-02-003

Explain decisions on edges

Structure and alignment

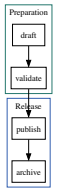
3 diagrams

Align parallel work, group related steps and connect to a specific table cell.



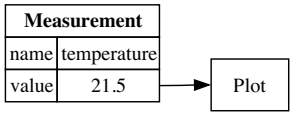
GVZ-03-001

Align parallel steps



GVZ-03-002

Group nodes into clusters



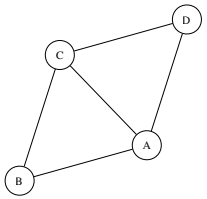
GVZ-03-003

Connect to a table cell

Layout engines

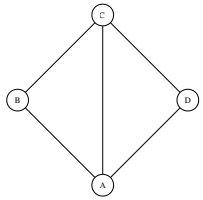
3 diagrams

Compare force-directed, circular and radial layouts for different relationships.



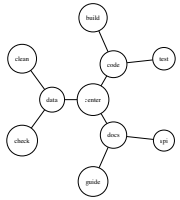
GVZ-04-001

Arrange a network with neato



GVZ-04-002

Arrange a cycle with circo



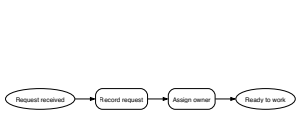
GVZ-04-003

Expand from a root with twopi

Process building blocks

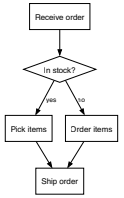
6 diagrams

Build a process from sequences, decisions, parallel tasks, approval gates and states.



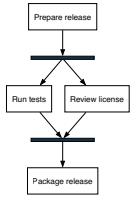
GVZ-05-001

Mark a clear start and finish



GVZ-05-002

Choose exactly one branch



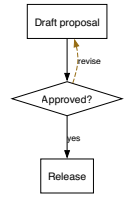
GVZ-05-003

Fork work and wait for both branches



GVZ-05-004

Show an optional activity



GVZ-05-005

Make approval a gate



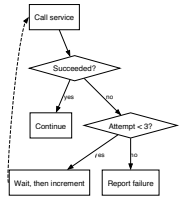
GVZ-05-006

Separate states from activities

Exceptions, retries and recovery

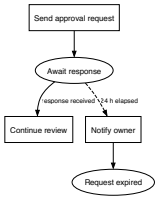
6 diagrams

Make retry limits, timeouts, fallbacks, compensation and failure exits visible.



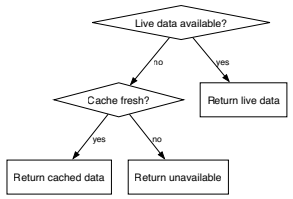
GVZ-06-001

Bound a retry loop



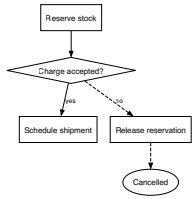
GVZ-06-002

Give a waiting step a timeout



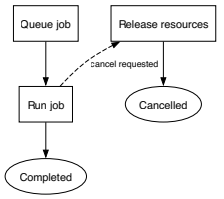
GVZ-06-003

Use a fallback with a declared limit



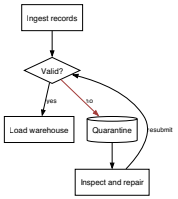
GVZ-06-004

Compensate for completed work



GVZ-06-005

Make cancellation a separate outcome



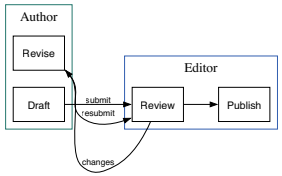
GVZ-06-006

Quarantine invalid input for repair

Roles, handoffs and swimlanes

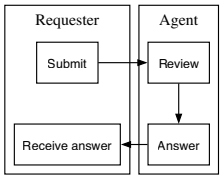
6 diagrams

Show who owns each step, where work changes hands and how escalation returns.



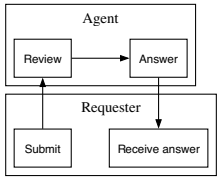
GVZ-07-001

Expose a handoff between teams



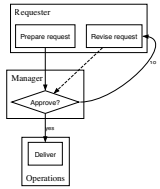
GVZ-07-002

Build two vertical swimlanes



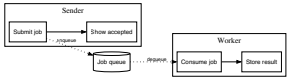
GVZ-07-003

Turn the lanes horizontally



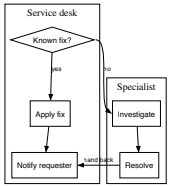
GVZ-07-004

Separate a decision from its owner



GVZ-07-005

Distinguish asynchronous messages



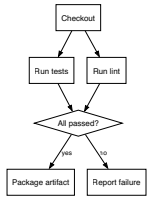
GVZ-07-006

Escalate with an explicit return path

Software and data workflows

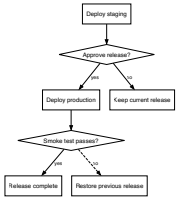
6 diagrams

Adapt complete patterns for CI, releases, data pipelines, queues and publishing.



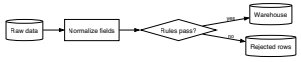
GVZ-08-001

Gate a build on independent checks



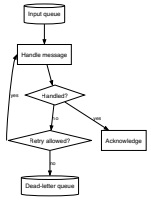
GVZ-08-002

Promote a release and verify it



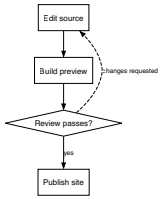
GVZ-08-003

Show both good and rejected data



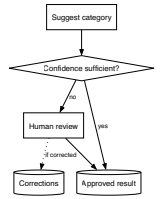
GVZ-08-004

Handle a poison message



GVZ-08-005

Publish documentation with review



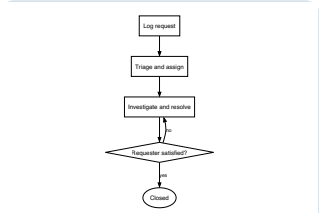
GVZ-08-006

Keep a human review in the loop

Service and operational processes

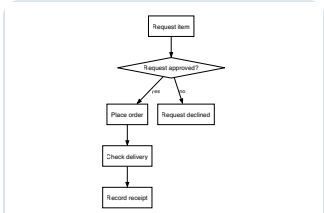
6 diagrams

Map service requests, purchasing, rework, hiring, onboarding and incident learning.



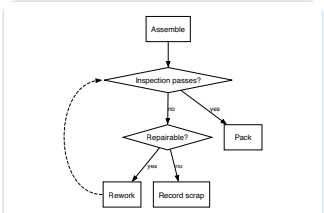
GVZ-09-001

Close a service request with confirmation



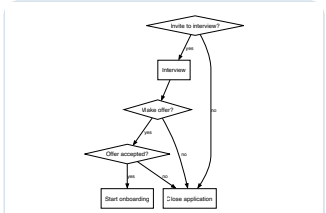
GVZ-09-002

Trace a purchase request to receipt



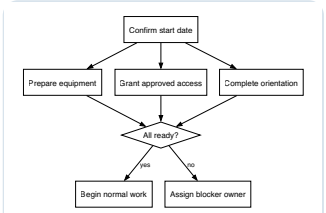
GVZ-09-003

Route defects through rework



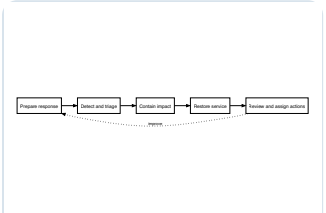
GVZ-09-004

Track a hiring workflow with explicit exits



GVZ-09-005

Coordinate onboarding prerequisites



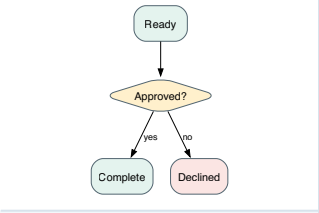
GVZ-09-006

Include learning after restoration

Visual language and design

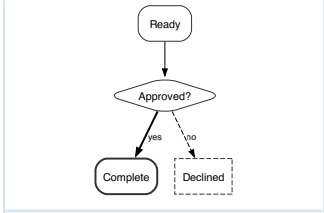
6 diagrams

Compare palettes and monochrome styles; tune emphasis, typography, padding and legends.



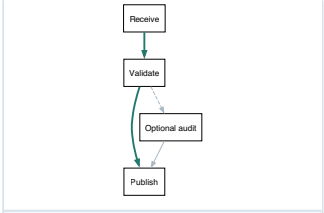
GVZ-10-001

Use a small semantic palette



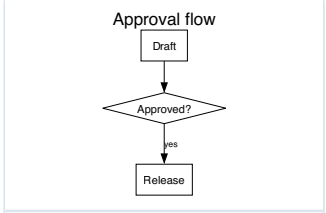
GVZ-10-002

Make the design work in monochrome



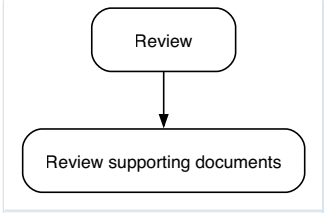
GVZ-10-003

Emphasize the path you want readers to follow



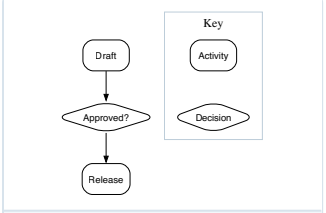
GVZ-10-004

Set typography at three levels



GVZ-10-005

Control padding and minimum node size



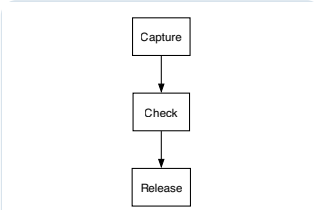
GVZ-10-006

Put the legend next to the process

Ranks, spacing and order

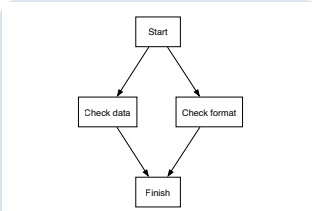
6 diagrams

Control reading direction, spacing, rank distances, branch order and alignment.



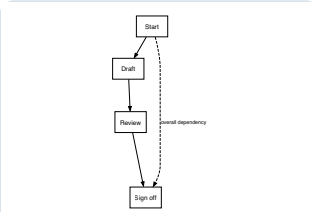
GVZ-11-001

Choose a reading direction



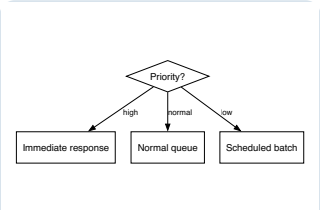
GVZ-11-002

Separate rank spacing from node spacing



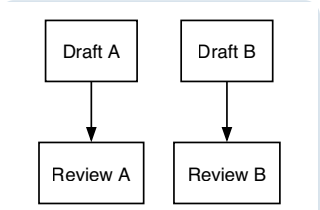
GVZ-11-003

Reserve space for a longer dependency



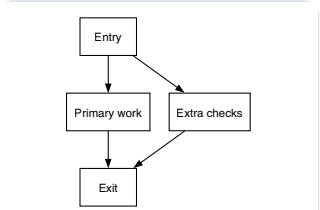
GVZ-11-004

Keep branch order stable



GVZ-11-005

Use invisible edges as layout scaffolding



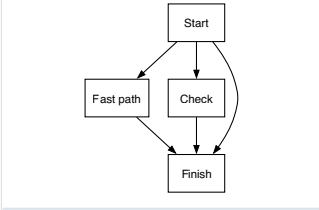
GVZ-11-006

Anchor the entry and exit ranks

Edge routing and connections

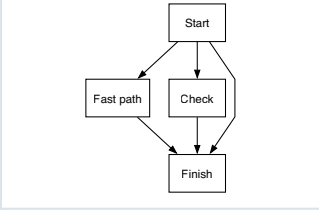
6 diagrams

Compare spline, polyline and orthogonal routes, then refine ports and feedback edges.



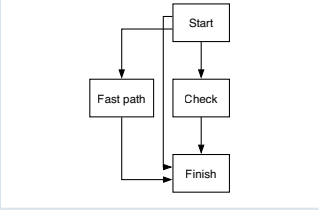
GVZ-12-001

Route smooth splines



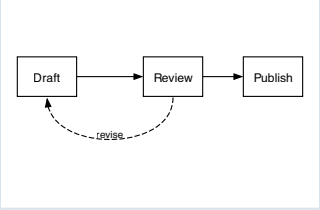
GVZ-12-002

Use segmented polylines



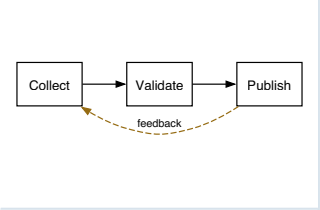
GVZ-12-003

Use orthogonal routing carefully



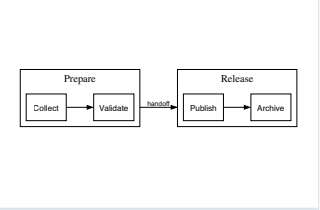
GVZ-12-004

Choose where edges meet a node



GVZ-12-005

Keep feedback from pulling the main flow



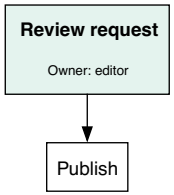
GVZ-12-006

Connect cluster boundaries

Rich labels and annotations

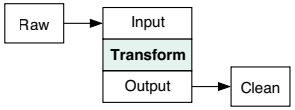
6 diagrams

Add structured cards, named ports, multiline text, endpoint labels and annotations.



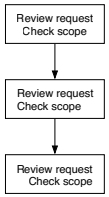
GVZ-13-001

Build an activity card with a subtitle



GVZ-13-002

Give inputs and outputs named ports



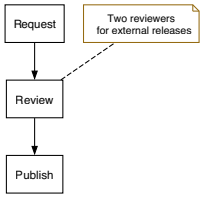
GVZ-13-003

Align multiline text deliberately



GVZ-13-004

Label the relationship and its endpoints



GVZ-13-005

Attach a note without changing the process



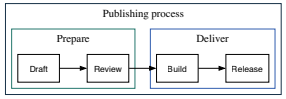
GVZ-13-006

Add a title and a scope statement

Composing larger process diagrams

6 diagrams

Combine phases, ownership, packing and visual design into complete process diagrams.



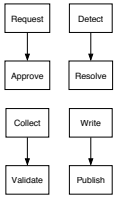
GVZ-14-001

Nest phases inside an overall process



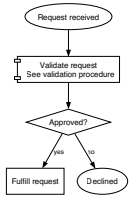
GVZ-14-002

Align three stages across two owners



GVZ-14-003

Pack independent workflows as small multiples



GVZ-14-004

Show detail through a named sub-process



GVZ-14-005

Compose a finished approval diagram



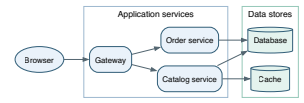
GVZ-14-006

Compose a process with artifacts and owners

Architecture and dependencies

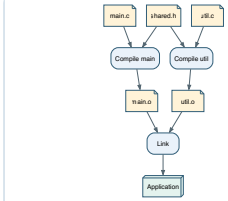
6 diagrams

Map services, build artifacts, data lineage, event distribution, replicas and module boundaries.



GVZ-15-001

Map a service architecture



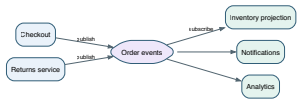
GVZ-15-002

Separate build actions from artifacts



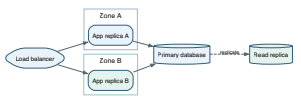
GVZ-15-003

Trace the lineage of a derived dataset



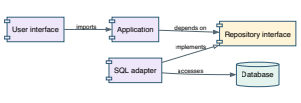
GVZ-15-004

Show event distribution to independent subscribers



GVZ-15-005

Distinguish replicas from independent services



GVZ-15-006

Draw dependencies around an interface

Trees and relationship maps

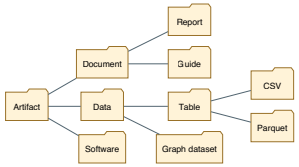
6 diagrams

Explore reporting lines, taxonomies, syntax trees, tries, radial maps and two-set relationships.



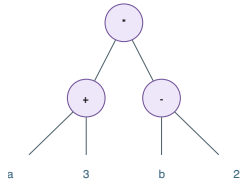
GVZ-16-001

Draw an organization tree



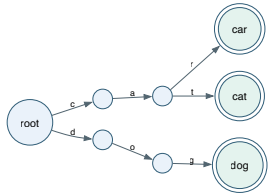
GVZ-16-002

Build a taxonomy with uneven branches



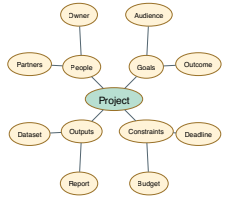
GVZ-16-003

Visualize an expression as a syntax tree



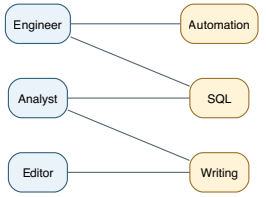
GVZ-16-004

Share prefixes in a trie



GVZ-16-005

Arrange a topic map around a central question



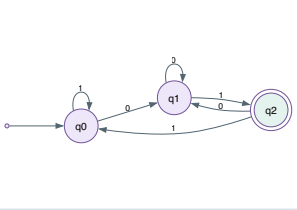
GVZ-16-006

Map relationships between two sets

Protocols and coordination

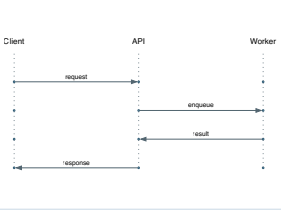
6 diagrams

Use automata, message ladders, Petri nets, circuit breakers, nested states and resource-wait graphs.



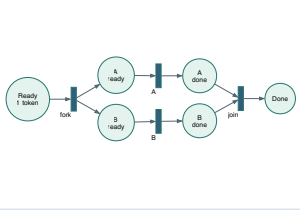
GVZ-17-001

Recognize strings with a finite automaton



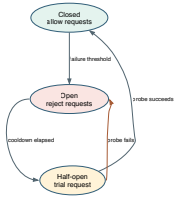
GVZ-17-002

Draw a message sequence as a ladder



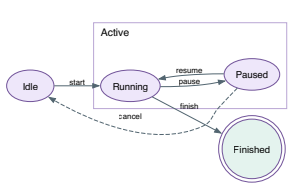
GVZ-17-003

Model a fork and join with a Petri net



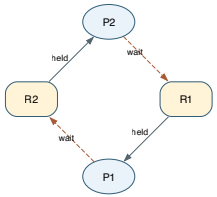
GVZ-17-004

Describe a circuit breaker as three states



GVZ-17-005

Group related states inside an active region



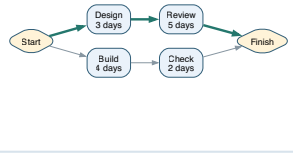
GVZ-17-006

Spot a circular resource wait

Planning and failure analysis

6 diagrams

Explain critical paths, milestones, fault trees, feedback, uncertain durations and intended impact.



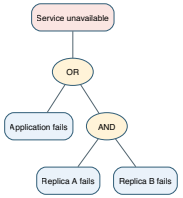
GVZ-18-001

Highlight a critical path through parallel work



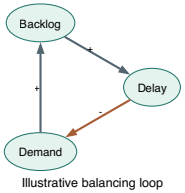
GVZ-18-002

Attach deliverables to a milestone roadmap



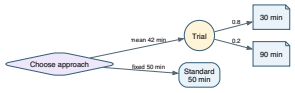
GVZ-18-003

Separate OR and AND conditions in a fault tree



GVZ-18-004

Draw a signed feedback loop



GVZ-18-005

Compare alternatives with uncertain durations



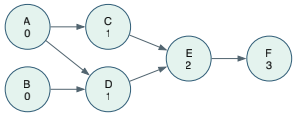
GVZ-18-006

Connect a goal to actors, behavior and deliverables

Algorithms and network reasoning

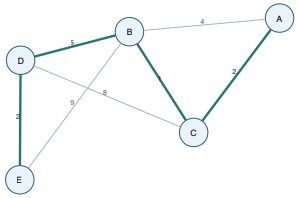
6 diagrams

Inspect topological layers, shortest paths, spanning trees, strongly connected components, coloring and flow.



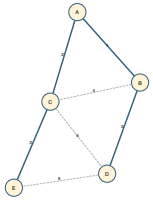
GVZ-19-001

Expose the layers of a dependency DAG



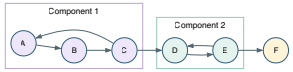
GVZ-19-002

Highlight a shortest route in a weighted network



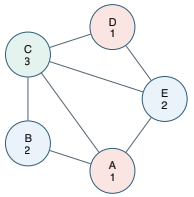
GVZ-19-003

Keep a minimum spanning tree inside the full network



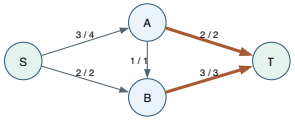
GVZ-19-004

Group mutually reachable vertices



GVZ-19-005

Color adjacent vertices differently



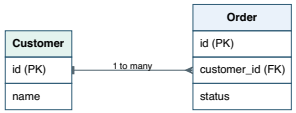
GVZ-19-006

Annotate a feasible maximum flow

Data structures and alternative layouts

6 diagrams

Connect database fields and pointer records, then explore fixed coordinates and force-directed networks.



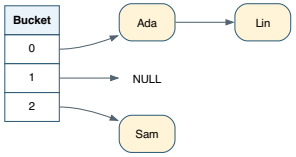
GVZ-20-001

Connect fields in an entity relationship diagram



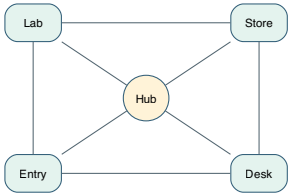
GVZ-20-002

Follow next pointers through a linked list



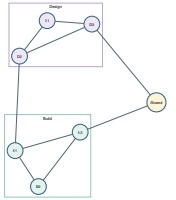
GVZ-20-003

Show collisions in a bucket table



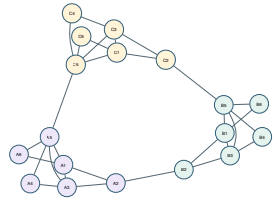
GVZ-20-004

Pin a spatial arrangement with neato



GVZ-20-005

Arrange clustered collaboration with fdp



GVZ-20-006

Reveal neighborhoods with sfdp

READING THE TOUR

Three ways to see a change

DOT source

Read a complete graph and distinguish process meaning from the attributes that arrange it.

Rendered diagram

Compare the source with the SVG produced by Graphviz itself.

Try a change

Use the Try changing prompts, follow comparison links and download complete DOT/SVG files.

Start with the image overview: select a diagram, or use search and the topic filter to find a process or design feature. Back to diagram overview returns to the same thumbnail. Each example includes its complete input and captured result. Copy link preserves the address of the guide you are reading, including its current domain and path.

Render a diagram

Graphviz turns a text description of nodes and edges into a diagram. DOT is the input language; `dot` is both a command and the default hierarchical layout engine. The examples specify another engine with `layout` when needed.

Download the DOT source from any example. From the folder containing that file:

```
dot -V
dot -Tsvg GVZ-01-001.dot -o diagram.svg
dot -Tpng GVZ-01-001.dot -o diagram.png
dot -Tpdf GVZ-01-001.dot -o diagram.pdf
```

`-T` selects the output format and `-o` names the file. SVG stays sharp when scaled; PNG is a raster image; PDF is useful in documents. Available renderers and fonts depend on the Graphviz installation. The tour prepares SVG figures; the PNG and PDF commands above are optional local exports.

To rebuild the whole guide, use `make full` in the tour folder. It explicitly runs reviewed DOT examples and then renders HTML/PDF. A subsequent `make` uses the prepared results. Each diagram starts from its own complete DOT source. HTML embeds its SVGs for offline reading; download links use the accompanying `examples/` and `figures/` folders. The guide PDF is produced by the shared publisher from the same prepared figures.

The examples are original, small demonstrations. The references below explain the language and attributes in more detail:

- DOT language and command line.
- Reading direction, rank constraints and edge constraints.
- Node shapes, HTML-like labels and ports.
- Selecting the engine, neato, circo and twopi.

Choose a reading path

The image overview above covers every example. Its first four chapters introduce DOT, followed by process patterns and design techniques. Choose a route below when you want to work through related chapters in order:

Goal	Chapters	Start here
Understand a process	05-06	Sequence, choices, parallel work and recovery
Make responsibilities visible	07	Handoffs, ownership and swimlanes
Adapt a practical workflow	08-09	Software, data, service and operational processes
Improve the visual design	10	Color, typography, emphasis and legends
Control placement and connections	11-12	Ranks, spacing, ports and routing
Add detail and compose a final diagram	13-14	Rich labels, annotations and complete recipes

Choose a process example close to your task, then use the design chapters to change its appearance. Each new example ends with a **Try changing** exercise. Comparison links connect variants that keep the process structure stable while changing a layout or style setting. All 108 examples remain independently runnable.

Process meaning and visual conventions

Graphviz lays out a graph; it does not execute activities, check business rules, schedule jobs, wait for both branches, or enforce a retry budget. Decide what nodes and arrows mean before tuning their appearance. These examples are illustrative process maps, not formal BPMN or UML conformance examples.

Visual element	Convention in these examples
Box with a verb	An activity or task
Diamond with a question	A decision; label every outgoing alternative
Ellipse	A start, finish, wait or state, as explained in the example
Narrow bar	A parallel fork or join; the text states the AND rule
Cylinder or note	Stored data or a document, not an activity
Cluster with a role name	Ownership boundary; crossings are handoffs
Dashed return arrow	Rework, retry or recovery, identified by its label
Invisible edge	Layout scaffolding only, never an implied business step

An exclusive merge and a parallel join have different meanings even when both have several incoming edges. State the rule explicitly. A long edge does not mean a long duration, and `weight` is not a task priority or elapsed time. Swimlanes here are constructed with clusters and rank constraints; dot does not provide a dedicated fixed-grid swimlane feature.

Chapters 15-20 broaden the gallery beyond workflow maps. Arrows may mean calls, dependencies, transitions, pointers or resource requests; each example states its own convention. Mathematical results and highlighted solutions are supplied in the source and explained in the text. Graphviz draws those results; it does not calculate schedules, run automata, simulate Petri nets or solve the graph optimization problems.

Design and layout reference

Start with short verb labels, one reading direction and an obvious outcome. Add branches and ownership boundaries next. Tune spacing before shrinking type, and inspect the result at the size where it will be used. Use color together with text, shapes or line styles so a monochrome version remains intelligible.

Control	What to adjust	Worked example
`rankdir`	Main reading direction in dot	Choose a direction
`ranksep` / `nodesep`	Between-rank / within-rank gaps in inches	Spacing
`rank` / `newrank`	Alignment, including across clusters	Aligned lanes
`minlen` / `weight`	Rank distance / preference for shorter, straighter edges	Dependency space
`ordering` / `group`	Branch order / straight connections within a group	Branch order
`constraint=false`	Exclude an edge from rank assignment	Feedback
`splines`	Curves, polylines or orthogonal routing	Routing comparison
Ports / `lhead` / `ltail`	Node attachment points / cluster boundaries	Cluster connections
`fontname` / `fontsize`	Requested font and point size	Typography
`margin` / `pad`	Node text inset / outer graph space	Padding
`color` / `fillcolor` / `penwidth`	Outline, interior and stroke thickness	Palette
HTML-like labels / `PORT`	Structured cards and named connections	Artifact ports

The `dot` engine is the default for the process chapters. Orthogonal routing has documented limitations with ports and ordinary edge labels; use the tested alternatives shown in chapter 12. Exact geometry can vary with fonts and Graphviz versions, even when the process structure is unchanged.

Primary references for process diagrams

These examples are original. The official documentation explains the Graphviz features used to draw them:

- DOT language, defaults, subgraphs and clusters.
- Shapes, HTML-like labels and port syntax.
- rankdir, rank, newrank, nodesep and ranksep.
- minlen, weight, ordering, group and constraint.
- Spline and orthogonal routing, compound, lhead and ltail.
- fixedsize, margin, pad and penwidth.
- headlabel, taillabel, xlabel, labeldistance and labelangle.
- Component packing and array packing modes.
- Arrow shapes and record fields and ports.
- Node positions, pinning and layout seeds.
- fdp clustered layouts and sfdp multilevel layouts.

First graphs

GVZ-01-001

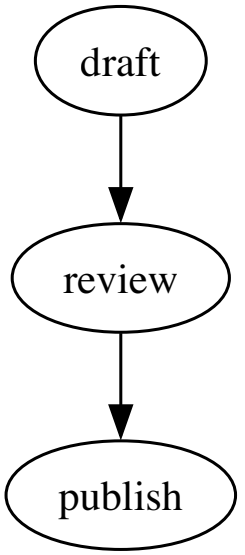
Connect three steps

A digraph describes directed relationships. Each arrow creates an edge; nodes are created when first mentioned. The chain expands to two edges. The default dot layout places the first step above the next.

DOT source

```
digraph workflow {
  draft -> review -> publish;
}
```

Actual output



GVZ-01-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-01-002

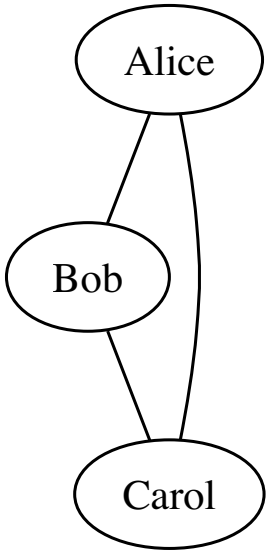
Draw an undirected network

Use graph and the double hyphen for relationships without a direction. This triangle has three nodes and three edges, with no arrowheads. Graphviz chooses positions from the connections; the source does not specify coordinates.

DOT source

```
graph team {
  Alice -- Bob;
  Bob -- Carol;
  Carol -- Alice;
}
```

Actual output



GVZ-01-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-01-003

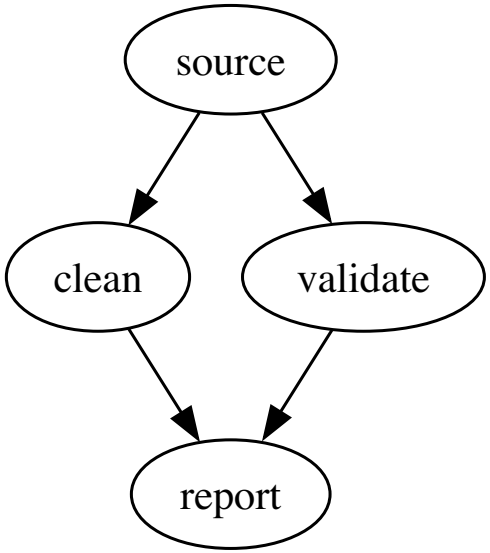
Branch and join

Braces collect nodes into an anonymous subgraph. The first statement connects source to both branches; the second connects both branches to report. Each name refers to the same node everywhere in the graph.

DOT source

```
digraph pipeline {
  source -> { clean validate };
  { clean validate } -> report;
}
```

Actual output



GVZ-01-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Labels and styling

GVZ-02-001

Separate names from labels

Keep short identifiers for connections and use label for readable text. The backslash-n inside a quoted label inserts a line break. rankdir=LR changes the dot layout to left-to-right; the node default makes subsequent nodes boxes.

DOT source

```
digraph workflow {
  rankdir=LR;
  node [shape=box];
  draft [label="Write\ndraft"];
  review [label="Peer review"];
  publish [label="Release"];
  draft -> review -> publish;
}
```

Actual output



GVZ-02-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-02-002

Set defaults and override a node

The node statement sets defaults for nodes created afterward. style combines rounded corners and a filled interior; fillcolor colors that interior and color controls the outline. A node-specific attribute overrides the default without changing its neighbors.

DOT source

```
digraph workflow {
  rankdir=LR;
  node [shape=box, style="rounded,filled",
        fillcolor="#e4f3ee", color="#24776d"];
  review [fillcolor="#fff0cc", color="#956b12"];
  draft -> review -> publish;
}
```

Actual output



GVZ-02-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-02-003

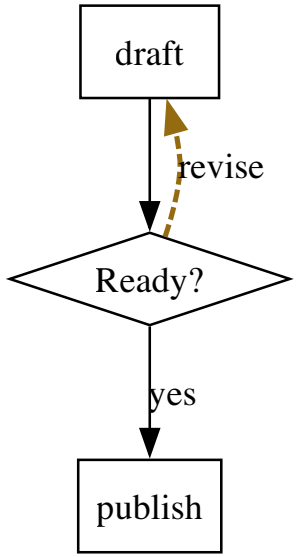
Explain decisions on edges

A diamond marks a decision while edge labels describe the branches. The dashed return path has a distinct color and penwidth. constraint=false prevents that edge from participating in dot rank assignment, keeping the retry path from defining the main flow.

DOT source

```
digraph decision {
  node [shape=box];
  check [shape=diamond, label="Ready?"];
  draft -> check;
  check -> publish [label="yes"];
  check -> draft [label="revise", style=dashed,
                  color="#956b12", penwidth=2,
                  constraint=false];
}
```

Actual output



GVZ-02-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Structure and alignment

GVZ-03-001

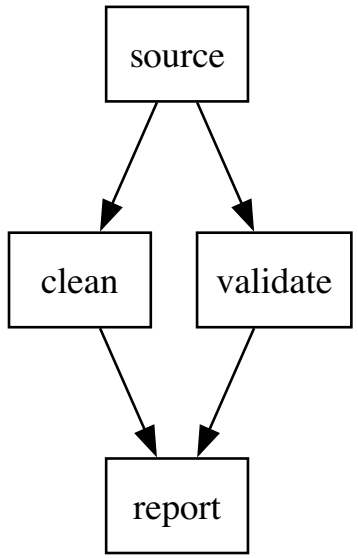
Align parallel steps

The rank=same subgraph puts clean and validate on the same level in dot. ranksep adjusts the space between levels, in inches. These are layout constraints: the edges still describe the same branch-and-join pipeline as GVZ-01-003.

DOT source

```
digraph pipeline {
  ranksep=0.7;
  node [shape=box];
  source -> clean -> report;
  source -> validate -> report;
  { rank=same; clean; validate; }
}
```

Actual output



GVZ-03-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-03-002

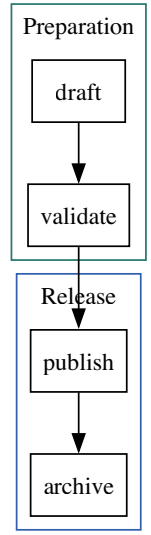
Group nodes into clusters

A subgraph whose name starts with cluster creates a visible group in dot. Each cluster gets its own label and border color. Nodes remain ordinary graph nodes, so an edge can connect validate to publish across the two groups.

DOT source

```
digraph delivery {
  node [shape=box];
  subgraph cluster_prepare {
    label="Preparation";
    color="#24776d";
    draft -> validate;
  }
  subgraph cluster_release {
    label="Release";
    color="#315eaf";
    publish -> archive;
  }
  validate -> publish;
}
```

Actual output



GVZ-03-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-03-003

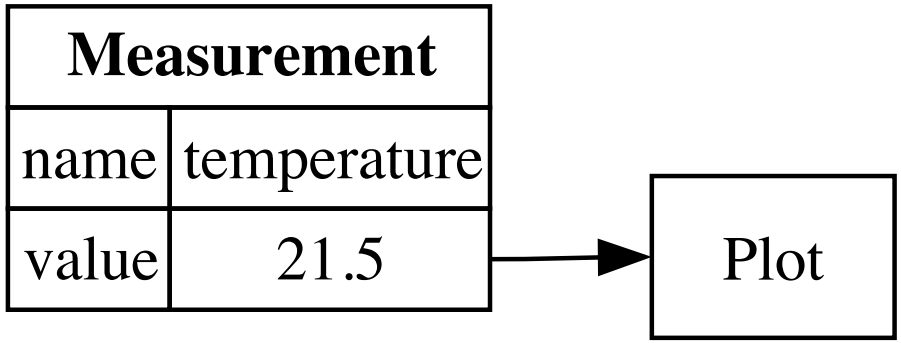
Connect to a table cell

Graphviz HTML-like labels use a supported subset of HTML, enclosed in angle brackets instead of quotes. shape=plain lets the table define the node boundary. PORT names a cell; fields:value:e targets that cell on its east side. BORDER and CELLBORDER control the table and cell outlines.

DOT source

```
digraph record {
  rankdir=LR;
  node [shape=plain];
  fields [label=<
    <TABLE BORDER="0" CELLBORDER="1" CELLSPACING="0">
      <TR><TD COLSPAN="2"><B>Measurement</B></TD></TR>
      <TR><TD>name</TD><TD>temperature</TD></TR>
      <TR><TD>value</TD><TD PORT="value">21.5</TD></TR>
    </TABLE>
  >];
  chart [shape=box, label="Plot"];
  fields:value:e -> chart:w;
}
```

Actual output



GVZ-03-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Layout engines

GVZ-04-001

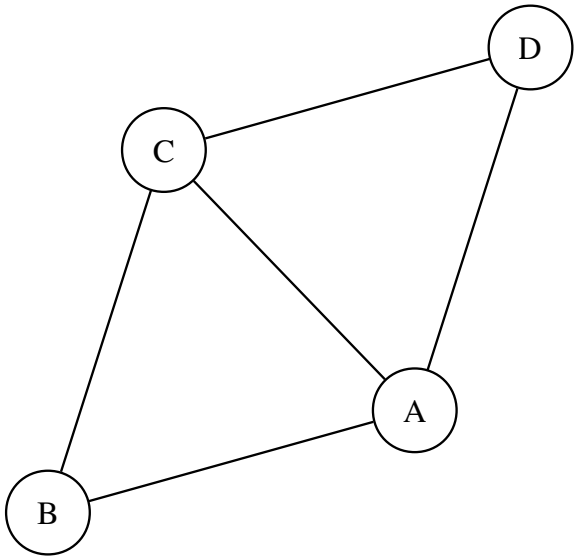
Arrange a network with neato

The layout attribute selects the engine inside the DOT source, so the same dot command can render every example. neato uses a spring-model layout. A fixed start value seeds its initial placement; overlap=false requests overlap removal. Exact positions can still vary with the Graphviz version and fonts.

DOT source

```
graph network {
  layout=neato;
  start=42;
  overlap=false;
  node [shape=circle];
  A -- B -- C -- D -- A;
  A -- C;
}
```

Actual output



GVZ-04-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-04-002

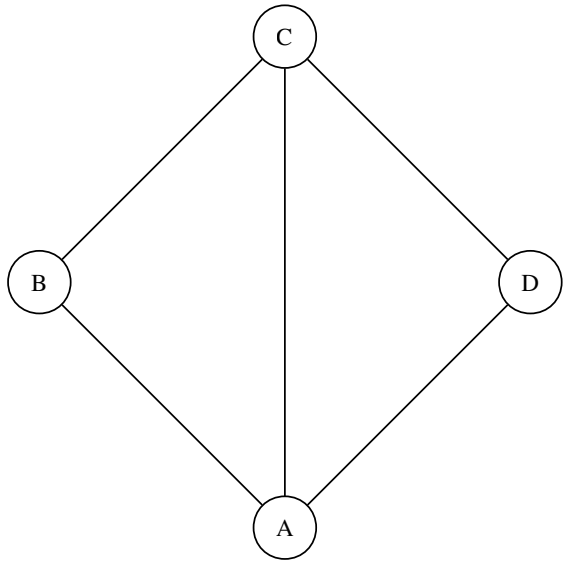
Arrange a cycle with circo

circo emphasizes cyclic structure. Compared with GVZ-04-001, this example changes only the layout engine; the network, seed and overlap setting remain in the source. The start seed is relevant to neato, while circo determines its circular ordering.

DOT source

```
graph network {
  layout=circo;
  start=42;
  overlap=false;
  node [shape=circle];
  A -- B -- C -- D -- A;
  A -- C;
}
```

Actual output



GVZ-04-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-04-003

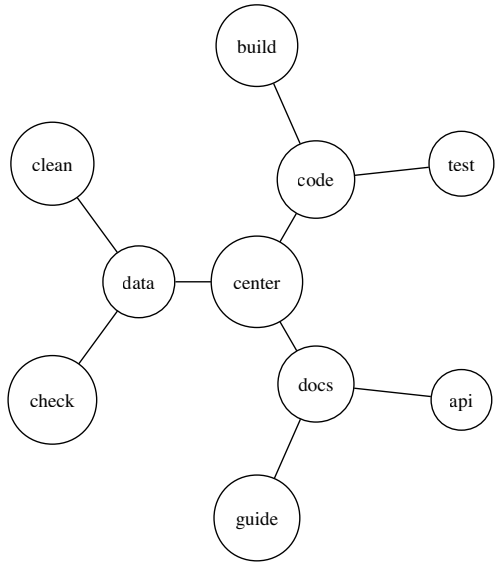
Expand from a root with twopi

twopi arranges nodes on concentric circles according to their graph distance from the root. root=center chooses the central node; ranksep sets the radial spacing in inches. Branch nodes sit one edge from center and leaves two edges away.

DOT source

```
graph knowledge {
  layout=twopi;
  root=center;
  ranksep=1.2;
  node [shape=circle];
  center -- { code data docs };
  code -- { test build };
  data -- { clean check };
  docs -- { guide api };
}
```

Actual output



GVZ-04-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Process building blocks

GVZ-05-001

Mark a clear start and finish

Make the boundary of a process visible before adding branches. Rounded boxes denote activities; ellipses mark the start and finish in this guide. The arrows describe sequence, while rankdir=LR chooses the reading direction. Naming activities with verbs helps readers distinguish work from outcomes.

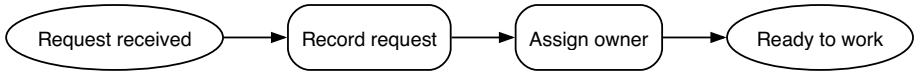
Try changing: Change LR to TB to compare a horizontal overview with a vertical checklist.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [style=rounded];
  start [shape=ellipse, label="Request received"];
  record [label="Record request"];
  assign [label="Assign owner"];
  finish [shape=ellipse, label="Ready to work"];
  start -> record -> assign -> finish;
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-05-001 · Rendered by Graphviz

Graphviz · exit 0

GVZ-05-002

Choose exactly one branch

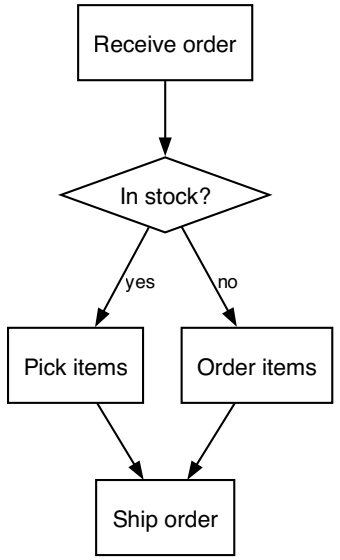
A diamond asks a question; yes and no label its alternatives. Both paths lead to the same next activity. This is an exclusive choice by convention, not a rule enforced by Graphviz. The merge does not wait for both branches because only one branch is intended to run.

Try changing: Rename the edge labels to the exact conditions used by your process.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  receive [label="Receive order"];
  stock [shape=diamond, label="In stock?"];
  pick [label="Pick items"];
  buy [label="Order items"];
  ship [label="Ship order"];
  receive -> stock;
  stock -> pick [label="yes"];
  stock -> buy [label="no"];
  { pick buy } -> ship;
}
```

Actual output



GVZ-05-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-05-003

Fork work and wait for both branches

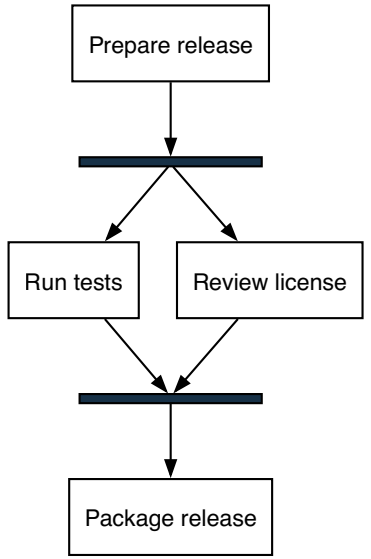
The narrow dark bars represent a parallel fork and a synchronization join. Packaging begins after both checks finish. Those AND semantics belong to the diagram convention; a pair of incoming arrows alone does not implement synchronization. Fixed dimensions are suitable for these deliberately empty bar labels.

Try changing: Add a third independent check between the same fork and join.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  start [label="Prepare release"];
  fork [shape=rect, label="", style=filled,
        fillcolor="#173147", width=1.2, height=0.06,
        fixedsize=true];
  join [shape=rect, label="", style=filled,
        fillcolor="#173147", width=1.2, height=0.06,
        fixedsize=true];
  test [label="Run tests"];
  audit [label="Review license"];
  package [label="Package release"];
  start -> fork;
  fork -> { test audit };
  { test audit } -> join;
  join -> package;
}
```

Actual output



GVZ-05-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-05-004

Show an optional activity

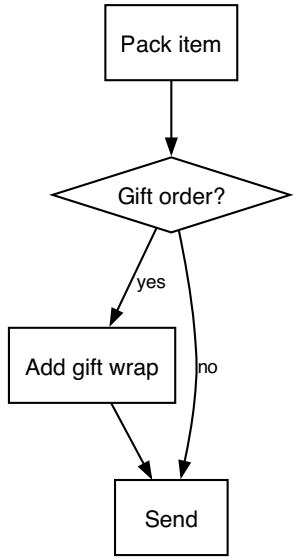
The optional activity has a visible bypass, so the diagram states what happens when it is skipped. Both paths converge at Send. A direct arrow from the decision to Send is clearer than a disconnected note saying that wrapping is optional.

Try changing: Give the optional branch a distinct label while keeping its destination explicit.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  pack [label="Pack item"];
  gift [shape=diamond, label="Gift order?"];
  wrap [label="Add gift wrap"];
  send [label="Send"];
  pack -> gift;
  gift -> wrap [label="yes"];
  gift -> send [label="no"];
  wrap -> send;
}
```

Actual output



GVZ-05-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-05-005

Make approval a gate

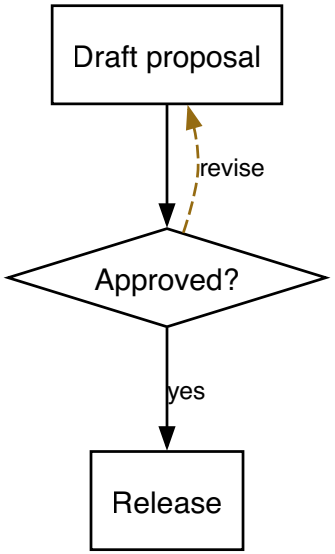
Approval is a decision with two distinct outcomes. A rejected proposal returns for revision; an approved one advances. The return edge uses constraint=false so it does not determine the forward ranks. Dashed styling distinguishes rework, but the revise label carries the meaning without color.

Try changing: Add a separate terminal rejection if some proposals must never return to drafting.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  draft [label="Draft proposal"];
  review [shape=diamond, label="Approved?"];
  release [label="Release"];
  draft -> review;
  review -> release [label="yes"];
  review -> draft [label="revise", style=dashed,
                   constraint=false, color="#956b12"];
}
```

Actual output



GVZ-05-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-05-006

Separate states from activities

This is a state-transition diagram: nodes name conditions and edges name events. It complements an activity flow without mixing the two vocabularies. A point marks the initial entry; doublecircle marks the terminal Closed state. The Reopened transition is an explicit exception.

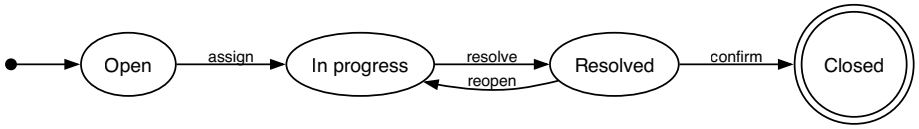
Try changing: Use verbs on transitions and stable condition names on states.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=ellipse];
  entry [shape=point, width=0.08];
  open [label="Open"];
  working [label="In progress"];
  resolved [label="Resolved"];
  closed [shape=doublecircle, label="Closed"];
  entry -> open;
  open -> working [label="assign"];
  working -> resolved [label="resolve"];
  resolved -> closed [label="confirm"];
  resolved -> working [label="reopen", constraint=false];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-05-006 · Rendered by Graphviz

Graphviz · exit 0

Exceptions, retries and recovery

GVZ-06-001

Bound a retry loop

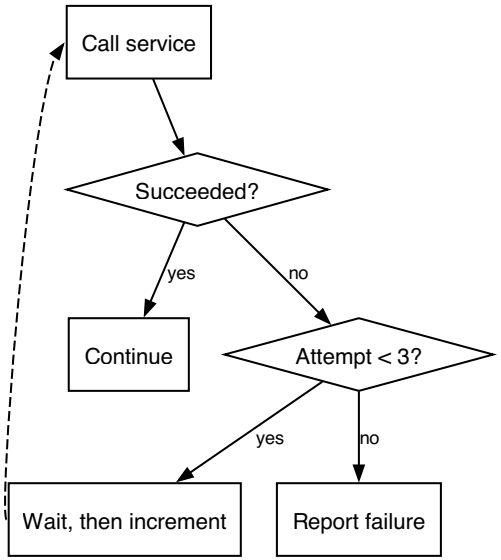
A retry needs both a success exit and a stopping condition. The second decision makes the attempt budget visible; the pause records backoff as an activity. This drawing documents the policy and does not count attempts or schedule waits. The actual retry implementation must enforce the same limit.

Try changing: Change the retry budget and name the error types that are eligible for retry.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  call [label="Call service"];
  ok [shape=diamond, label="Succeeded?"];
  budget [shape=diamond, label="Attempt < 3?"];
  wait [label="Wait, then increment"];
  done [label="Continue"];
  fail [label="Report failure"];
  call -> ok;
  ok -> done [label="yes"];
  ok -> budget [label="no"];
  budget -> wait [label="yes"];
  budget -> fail [label="no"];
  wait:w -> call:w [style=dashed];
}
```

Actual output



GVZ-06-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-06-002

Give a waiting step a timeout

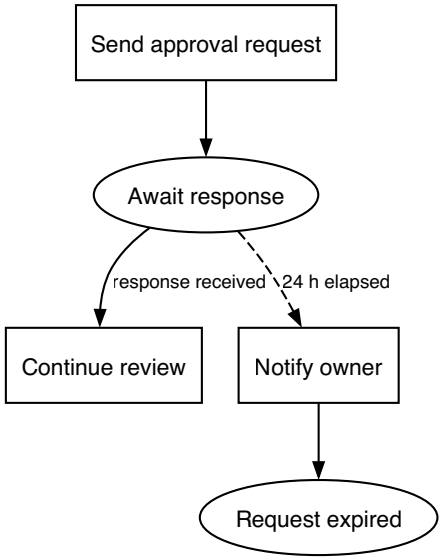
Waiting has two competing outcomes: a reply or an expired deadline. The edge labels state which event advances the process. A timeout path ends explicitly instead of hanging in mid-air. The diagram does not express a duration by geometric distance; the 24 h label is the source of that information.

Try changing: Replace the example deadline with a named business rule rather than a longer arrow.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  send [label="Send approval request"];
  wait [shape=ellipse, label="Await response"];
  accepted [label="Continue review"];
  expired [label="Notify owner"];
  stop [shape=ellipse, label="Request expired"];
  send -> wait;
  wait -> accepted [label="response received"];
  wait -> expired [label="24 h elapsed", style=dashed];
  expired -> stop;
}
```

Actual output



GVZ-06-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-06-003

Use a fallback with a declared limit

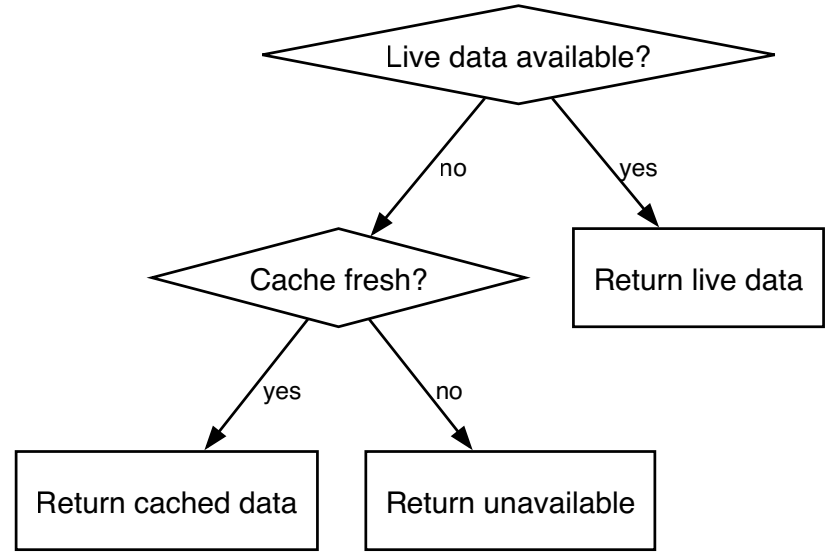
A failed primary lookup can use a cached response, but only if that response is fresh enough. Both branches end in an explicit outcome. This separates graceful degradation from silent success and makes the fallback condition reviewable.

Try changing: Label the cached result explicitly so consumers can distinguish it from live data.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  live [shape=diamond, label="Live data available?"];
  cache [shape=diamond, label="Cache fresh?"];
  current [label="Return live data"];
  stale [label="Return cached data"];
  error [label="Return unavailable"];
  live -> current [label="yes"];
  live -> cache [label="no"];
  cache -> stale [label="yes"];
  cache -> error [label="no"];
}
```

Actual output



GVZ-06-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-06-004

Compensate for completed work

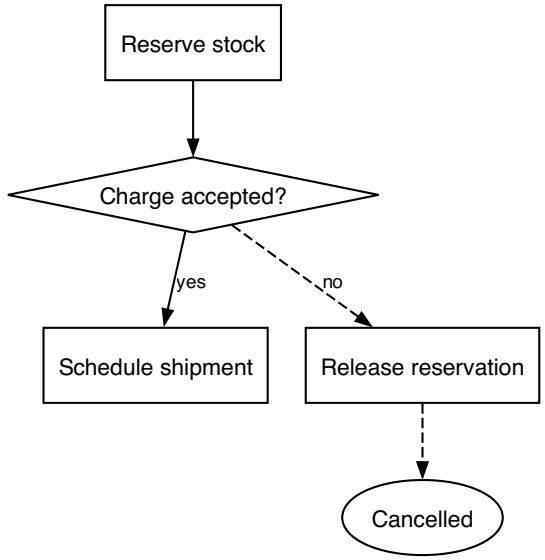
A later failure can require undoing earlier business actions. Here a failed charge releases a reservation. Compensation is a separate action, not a reversed normal-flow arrow: it may itself fail and may need monitoring in a larger design. Dashed edges identify the recovery path.

Try changing: Add a manual-resolution branch for a failed compensation in a larger process.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  reserve [label="Reserve stock"];
  charge [shape=diamond, label="Charge accepted?"];
  ship [label="Schedule shipment"];
  release [label="Release reservation"];
  cancelled [shape=ellipse, label="Cancelled"];
  reserve -> charge;
  charge -> ship [label="yes"];
  charge -> release [label="no", style=dashed];
  release -> cancelled [style=dashed];
}
```

Actual output



GVZ-06-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-06-005

Make cancellation a separate outcome

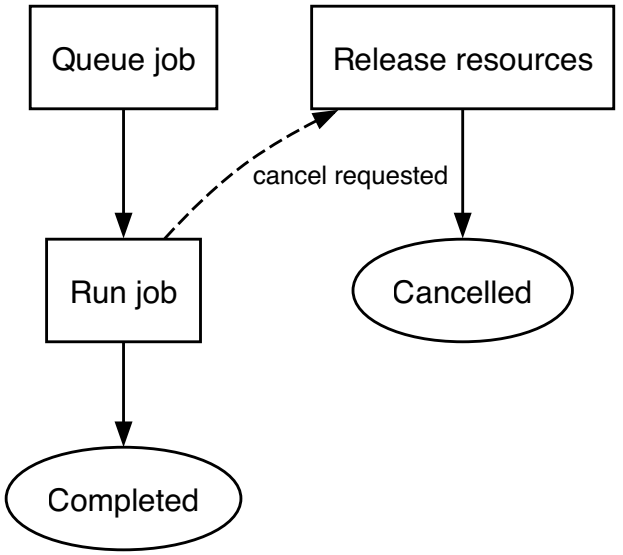
Cancellation leaves the ordinary completion path but still performs cleanup. The two exits are named, so a reader can distinguish completed work from interrupted work. The request-cancel edge is unconstrained to keep the main sequence easy to scan.

Try changing: Decide which activities permit cancellation and draw only those entry points.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  queue [label="Queue job"];
  run [label="Run job"];
  complete [shape=ellipse, label="Completed"];
  cleanup [label="Release resources"];
  cancel [shape=ellipse, label="Cancelled"];
  queue -> run -> complete;
  run -> cleanup [label="cancel requested",
                  style=dashed, constraint=false];
  cleanup -> cancel;
}
```

Actual output



GVZ-06-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-06-006

Quarantine invalid input for repair

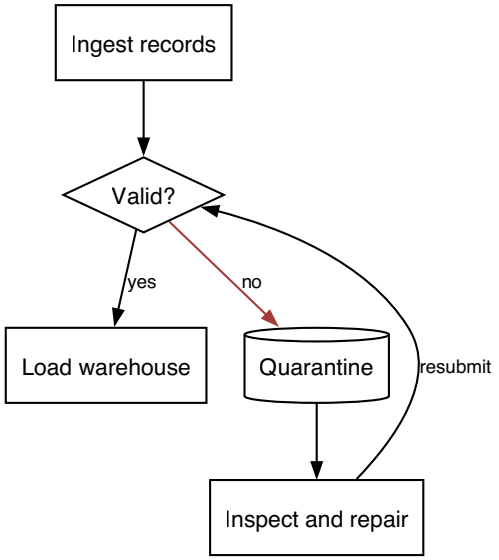
Invalid records are retained for inspection rather than discarded invisibly. The repair loop returns to validation, ensuring that repaired input follows the same gate as new input. A cylinder denotes retained data by this guide's convention; it does not create a database.

Try changing: Add a terminal discard route if some records cannot be repaired.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  ingest [label="Ingest records"];
  validate [shape=diamond, label="Valid?"];
  load [label="Load warehouse"];
  quarantine [shape=cylinder, label="Quarantine"];
  repair [label="Inspect and repair"];
  ingest -> validate;
  validate -> load [label="yes"];
  validate -> quarantine [label="no", color="#a33b38"];
  quarantine -> repair;
  repair -> validate [label="resubmit", constraint=false];
}
```

Actual output



GVZ-06-006 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Roles, handoffs and swimlanes

GVZ-07-001

Expose a handoff between teams

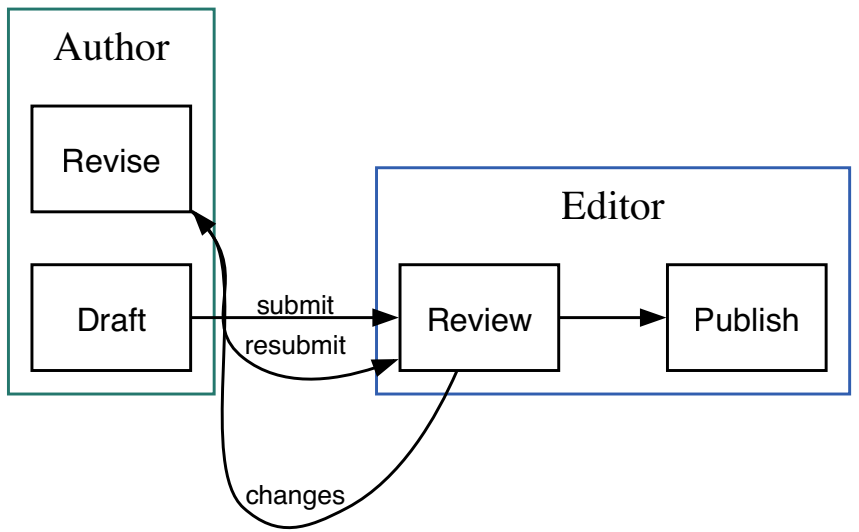
Clusters group activities by owner. The crossing arrow is the handoff: authors submit a draft to an editor, who reviews and publishes it. Graphviz clusters are containers for layout and styling; their names must start with cluster to receive the familiar boundary treatment.

Try changing: Rename each cluster to the role accountable for its activities.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  subgraph cluster_author {
    label="Author"; color="#24776d";
    draft [label="Draft"];
    revise [label="Revise"];
  }
  subgraph cluster_editor {
    label="Editor"; color="#315eaf";
    review [label="Review"];
    publish [label="Publish"];
    review -> publish;
  }
  draft -> review [label="submit"];
  review -> revise [label="changes", constraint=false];
  revise -> review [label="resubmit", constraint=false];
}
```

Actual output



GVZ-07-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-07-002

Build two vertical swimlanes

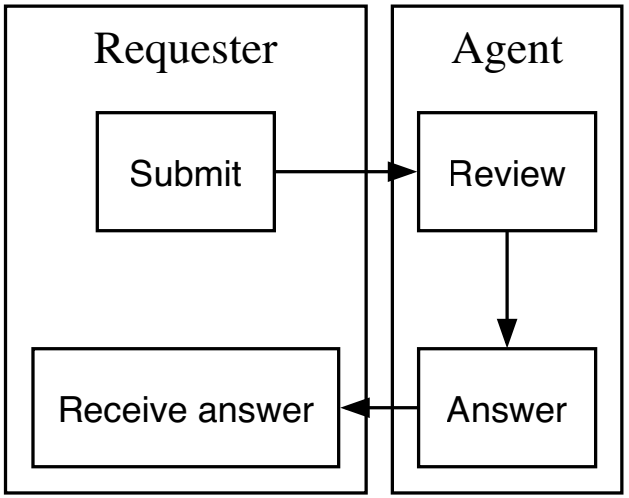
The lanes use clusters for ownership, same-rank constraints for shared stages, and newrank=true to rank across cluster boundaries. Invisible edges align successive positions without claiming an extra business handoff. Dot computes the geometry; this is an approximate swimlane layout, not a fixed table.

Try changing: Remove the invisible edge to see how the layout loses its lane scaffold.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  newrank=true;
  subgraph cluster_requester {
    label="Requester";
    ask [label="Submit"];
    receive [label="Receive answer"];
    ask -> receive [style=invis];
  }
  subgraph cluster_agent {
    label="Agent";
    review [label="Review"];
    answer [label="Answer"];
    review -> answer;
  }
  { rank=same; ask; review; }
  { rank=same; receive; answer; }
  ask -> review [constraint=false];
  answer -> receive [constraint=false];
}
```

Actual output



GVZ-07-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-07-003

Turn the lanes horizontally

This uses the same ownership, rank constraints and edges as GVZ-07-002. Changing only rankdir rotates the progression: same-rank pairs now share a vertical column and each role progresses left to right. A lane orientation is a reading choice, not a change to process semantics.

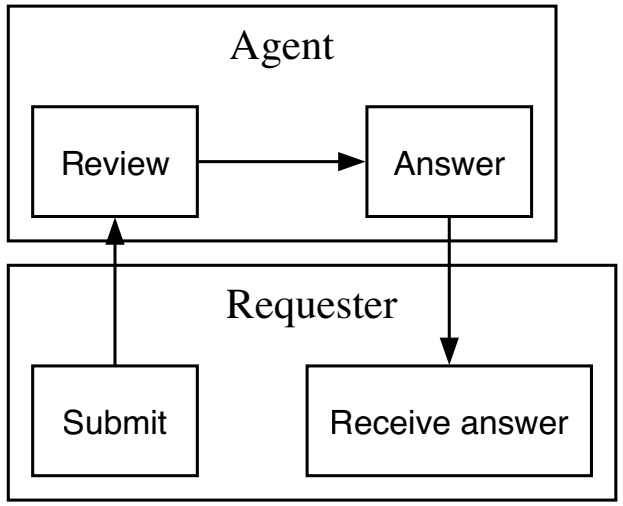
Try changing: Compare both orientations at the width where the final diagram will be read.

Compare with GVZ-07-002 · Build two vertical swimlanes.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  newrank=true;
  subgraph cluster_requester {
    label="Requester";
    ask [label="Submit"];
    receive [label="Receive answer"];
    ask -> receive [style=invis];
  }
  subgraph cluster_agent {
    label="Agent";
    review [label="Review"];
    answer [label="Answer"];
    review -> answer;
  }
  { rank=same; ask; review; }
  { rank=same; receive; answer; }
  ask -> review [constraint=false];
  answer -> receive [constraint=false];
}
```

Actual output



GVZ-07-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-07-004

Separate a decision from its owner

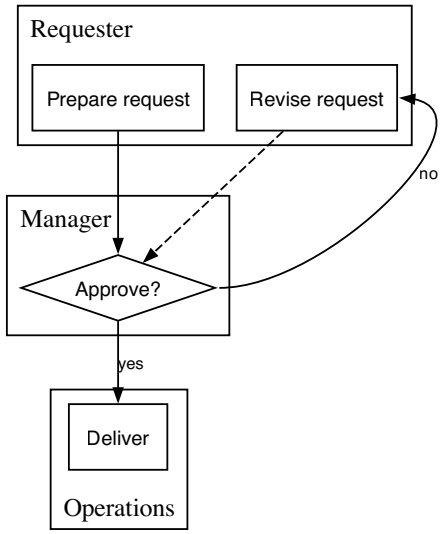
The requester prepares the proposal, the manager makes the decision, and operations delivers an approved request. Keeping the decision inside the manager cluster makes authority visible. The rejected path returns to the requester; an approval does not move ownership of the proposal itself.

Try changing: Use role names instead of individual names when the process should survive staff changes.

DOT source

```
digraph process {
  rankdir=TB;
  labeljust=l;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  subgraph cluster_requester {
    label="Requester";
    prepare [label="Prepare request"];
    revise [label="Revise request"];
  }
  subgraph cluster_manager {
    label="Manager";
    approve [shape=diamond, label="Approve?"];
  }
  subgraph cluster_ops {
    label="Operations"; labelloc=b;
    deliver [label="Deliver"];
  }
  prepare -> approve;
  approve -> deliver [label="yes"];
  approve:e -> revise:e [label="no", constraint=false];
  revise -> approve [constraint=false, style=dashed];
}
```

Actual output



GVZ-07-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-07-005

Distinguish asynchronous messages

The sender posts a message and can continue before the receiver finishes. A dotted edge denotes message transport in this example; solid arrows denote local sequence. The queue separates ownership and timing, while the explanation prevents readers from mistaking the picture for an executable messaging specification.

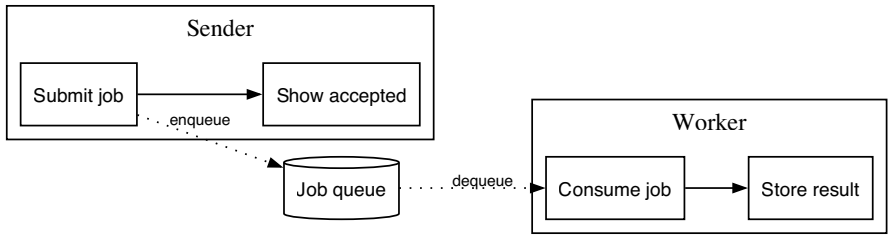
Try changing: Add correlation or acknowledgment labels when the handoff needs more precision.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  subgraph cluster_sender {
    label="Sender";
    submit [label="Submit job"];
    ack [label="Show accepted"];
    submit -> ack;
  }
  queue [shape=cylinder, label="Job queue"];
  subgraph cluster_worker {
    label="Worker";
    consume [label="Consume job"];
    finish [label="Store result"];
    consume -> finish;
  }
  submit -> queue [style=dotted, label="enqueue"];
  queue -> consume [style=dotted, label="dequeue"];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-07-005 · Rendered by Graphviz

Graphviz · exit 0

GVZ-07-006

Escalate with an explicit return path

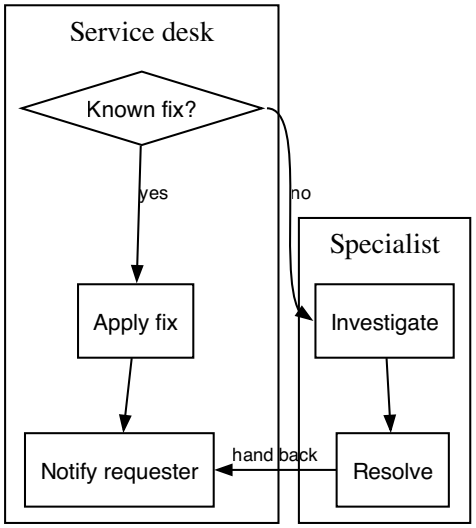
An escalation changes who handles a case, then returns the result to the original owner. The specialist is a separate cluster rather than just a different-colored node. The final notification remains the service desk's responsibility, making the handback as clear as the initial handoff.

Try changing: State the escalation condition on the crossing edge, not only in a separate legend.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  subgraph cluster_desk {
    label="Service desk";
    triage [shape=diamond, label="Known fix?"];
    fix [label="Apply fix"];
    notify [label="Notify requester"];
    triage -> fix [label="yes"];
    fix -> notify;
  }
  subgraph cluster_specialist {
    label="Specialist";
    investigate [label="Investigate"];
    resolve [label="Resolve"];
    investigate -> resolve;
  }
  triage:e -> investigate:w [label="no"];
  resolve -> notify [label="hand back"];
}
```

Actual output



GVZ-07-006 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Software and data workflows

GVZ-08-001

Gate a build on independent checks

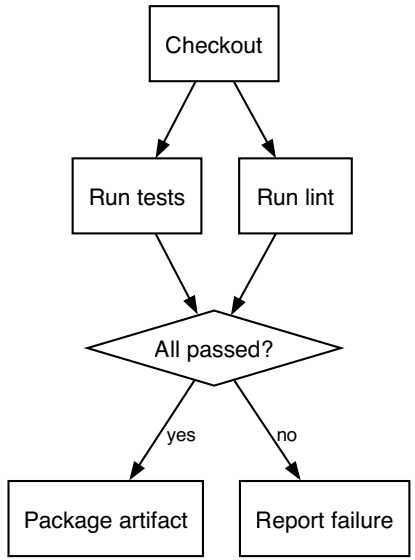
Tests and linting can run independently after checkout. The diamond names the all-pass rule before an artifact is packaged. This documents a dependency and decision structure; incoming edges do not make Graphviz evaluate results. The failure outcome is drawn instead of omitted.

Try changing: Add security or license checks as named prerequisites of the same gate.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  checkout [label="Checkout"];
  tests [label="Run tests"];
  lint [label="Run lint"];
  gate [shape=diamond, label="All passed?"];
  package [label="Package artifact"];
  fail [label="Report failure"];
  checkout -> { tests lint };
  { tests lint } -> gate;
  gate -> package [label="yes"];
  gate -> fail [label="no"];
}
```

Actual output



GVZ-08-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-08-002

Promote a release and verify it

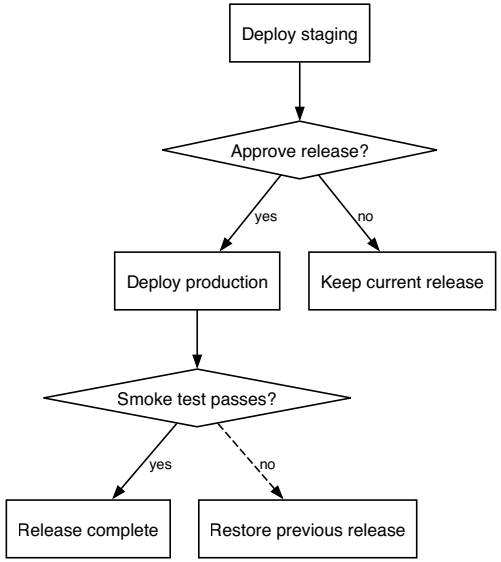
Deployment continues only after a manual approval. A smoke-test failure triggers rollback to the previous release. Distinct failure and success ends make this a deployment process rather than a picture of environments alone. The dashed rollback route is deliberately secondary in the layout.

Try changing: Name the release being restored; a generic failure arrow is not a rollback plan.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  stage [label="Deploy staging"];
  approve [shape=diamond, label="Approve release?"];
  prod [label="Deploy production"];
  smoke [shape=diamond, label="Smoke test passes?"];
  done [label="Release complete"];
  hold [label="Keep current release"];
  rollback [label="Restore previous release"];
  stage -> approve;
  approve -> prod [label="yes"];
  approve -> hold [label="no"];
  prod -> smoke;
  smoke -> done [label="yes"];
  smoke -> rollback [label="no", style=dashed];
}
```

Actual output



GVZ-08-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-08-003

Show both good and rejected data

An ETL flow distinguishes stored datasets from transforming activities. Cylinders denote retained data, boxes denote work, and the diamond is a quality decision. Rejected rows have their own destination. Shapes and labels preserve the distinctions when the drawing is printed without color.

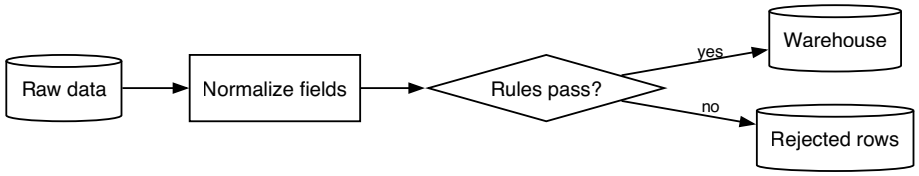
Try changing: Give each transformation a verb and each dataset a noun.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  raw [shape=cylinder, label="Raw data"];
  transform [label="Normalize fields"];
  quality [shape=diamond, label="Rules pass?"];
  warehouse [shape=cylinder, label="Warehouse"];
  rejected [shape=cylinder, label="Rejected rows"];
  raw -> transform -> quality;
  quality -> warehouse [label="yes"];
  quality -> rejected [label="no"];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-08-003 · Rendered by Graphviz

Graphviz · exit 0

GVZ-08-004

Handle a poison message

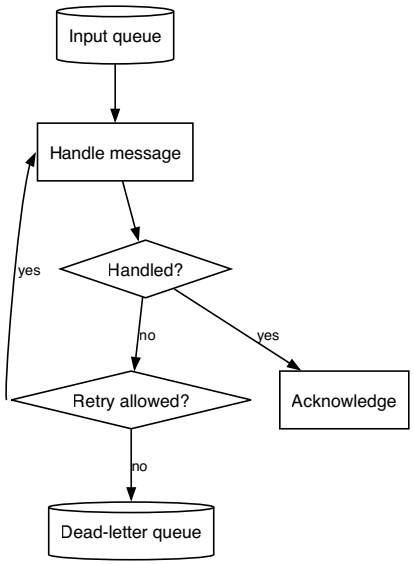
A consumer tries a message, records success, or routes repeated failure to a dead-letter queue. The retry counter is part of the application policy, shown explicitly as a condition. Routing failures away from the main queue prevents the process description from implying unbounded retries.

Try changing: Add an inspection and replay process for the dead-letter queue as a separate diagram.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  queue [shape=cylinder, label="Input queue"];
  handle [label="Handle message"];
  ok [shape=diamond, label="Handled?"];
  retry [shape=diamond, label="Retry allowed?"];
  ack [label="Acknowledge"];
  dead [shape=cylinder, label="Dead-letter queue"];
  queue -> handle -> ok;
  ok -> ack [label="yes"];
  ok -> retry [label="no"];
  retry:w -> handle:w [label="yes"];
  retry -> dead [label="no"];
}
```

Actual output



GVZ-08-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-08-005

Publish documentation with review

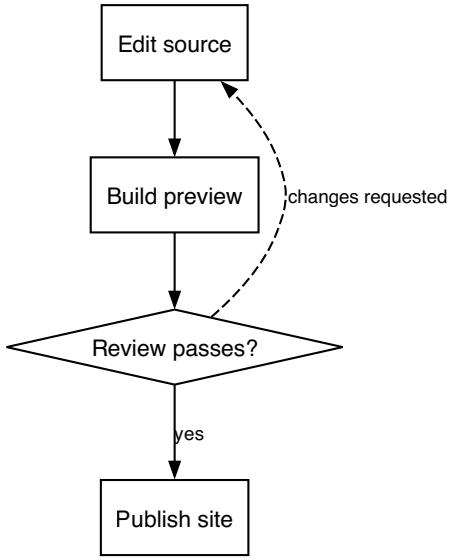
Source changes produce a preview that a person reviews before publication. A failed review returns to editing. The diagram separates artifact production from approval and public release, making it suitable for documentation, reports or other generated material.

Try changing: Add a link-check activity before review when broken references are a release gate.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  edit [label="Edit source"];
  build [label="Build preview"];
  review [shape=diamond, label="Review passes?"];
  publish [label="Publish site"];
  edit -> build -> review;
  review -> publish [label="yes"];
  review -> edit [label="changes requested",
                  constraint=false, style=dashed];
}
```

Actual output



GVZ-08-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-08-006

Keep a human review in the loop

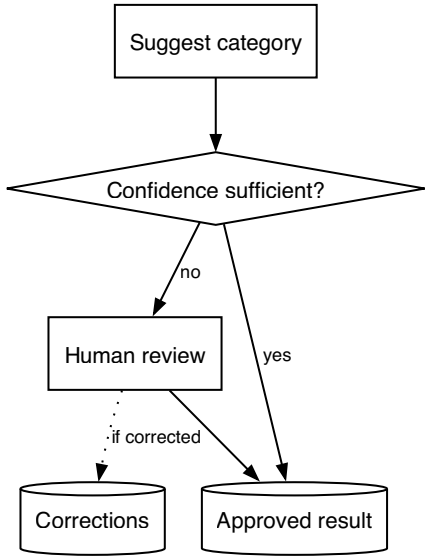
Automation proposes a classification and a person checks uncertain cases. The confidence threshold is a policy input, not a property that Graphviz computes. Both accepted routes converge on the same saved result, while a correction is recorded as feedback for later improvement.

Try changing: Describe which cases always require review, regardless of the score.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  suggest [label="Suggest category"];
  confident [shape=diamond, label="Confidence sufficient?"];
  review [label="Human review"];
  save [shape=cylinder, label="Approved result"];
  feedback [shape=cylinder, label="Corrections"];
  suggest -> confident;
  confident -> save [label="yes"];
  confident -> review [label="no"];
  review -> save;
  review -> feedback [style=dotted, label="if corrected"];
}
```

Actual output



GVZ-08-006 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Service and operational processes

GVZ-09-001

Close a service request with confirmation

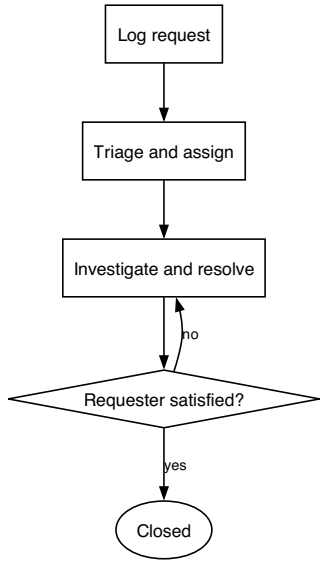
Resolution and closure are separate steps. The requester confirms that the solution worked; otherwise the case returns for investigation. This prevents a straight-line drawing from hiding the possibility of reopening. The unconstrained return edge preserves the main reading order.

Try changing: Make the ownership of confirmation explicit when closure requires a different role.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  log [label="Log request"];
  triage [label="Triage and assign"];
  solve [label="Investigate and resolve"];
  confirm [shape=diamond, label="Requester satisfied?"];
  close [shape=ellipse, label="Closed"];
  log -> triage -> solve -> confirm;
  confirm -> close [label="yes"];
  confirm -> solve [label="no", constraint=false];
}
```

Actual output



GVZ-09-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-09-002

Trace a purchase request to receipt

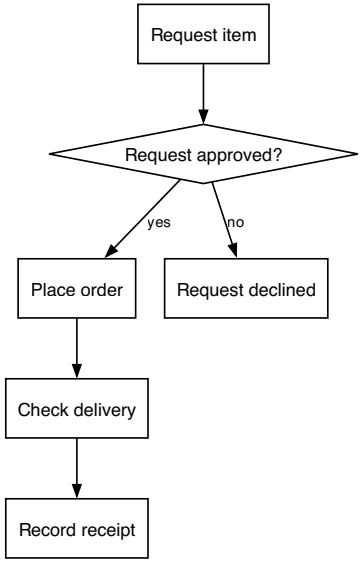
This illustrative purchasing flow separates approval, ordering and receipt. Rejection ends the request before an order is placed. It is a process example, not an organizational policy; actual approval thresholds and controls belong in the accompanying rules.

Try changing: Add a delivery-mismatch branch instead of assuming every received item is correct.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  request [label="Request item"];
  approve [shape=diamond, label="Request approved?"];
  order [label="Place order"];
  receive [label="Check delivery"];
  record [label="Record receipt"];
  stop [label="Request declined"];
  request -> approve;
  approve -> order [label="yes"];
  approve -> stop [label="no"];
  order -> receive -> record;
}
```

Actual output



GVZ-09-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-09-003

Route defects through rework

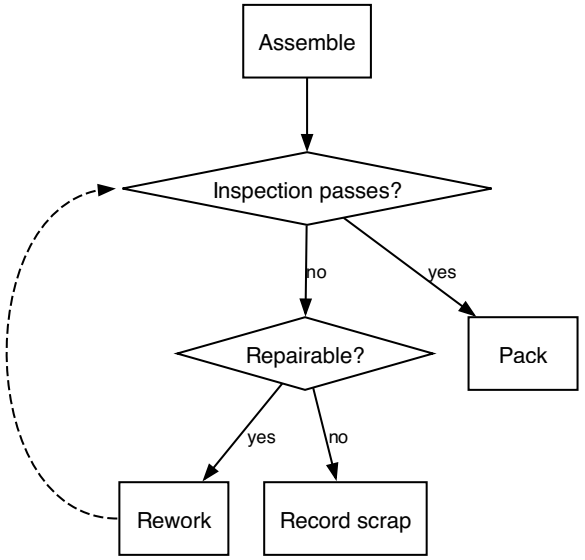
An inspection gate separates acceptable output from defects. A second question distinguishes repairable defects from scrap. Returning reworked items to inspection makes the quality loop explicit and avoids implying that a repair automatically passes.

Try changing: Name the inspection criterion and the maximum rework policy in the process documentation.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  build [label="Assemble"];
  inspect [shape=diamond, label="Inspection passes?"];
  pack [label="Pack"];
  repairable [shape=diamond, label="Repairable?"];
  rework [label="Rework"];
  scrap [label="Record scrap"];
  build -> inspect;
  inspect -> pack [label="yes"];
  inspect -> repairable [label="no"];
  repairable -> rework [label="yes"];
  repairable -> scrap [label="no"];
  rework:w -> inspect:w [style=dashed];
}
```

Actual output



GVZ-09-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-09-004

Track a hiring workflow with explicit exits

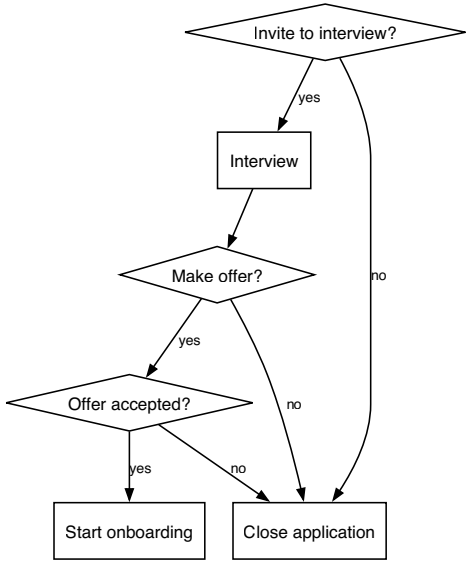
Applications move through review, interview and an offer decision. Unsuccessful and declined outcomes remain visible rather than disappearing from the chart. This example models stages only; it does not define candidate selection criteria or automate hiring decisions.

Try changing: Keep the stage diagram separate from the detailed evaluation rubric.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  review [shape=diamond, label="Invite to interview?"];
  interview [label="Interview"];
  offer [shape=diamond, label="Make offer?"];
  accept [shape=diamond, label="Offer accepted?"];
  onboard [label="Start onboarding"];
  end [label="Close application"];
  review -> interview [label="yes"];
  review -> end [label="no"];
  interview -> offer;
  offer -> accept [label="yes"];
  offer -> end [label="no"];
  accept -> onboard [label="yes"];
  accept -> end [label="no"];
}
```

Actual output



GVZ-09-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-09-005

Coordinate onboarding prerequisites

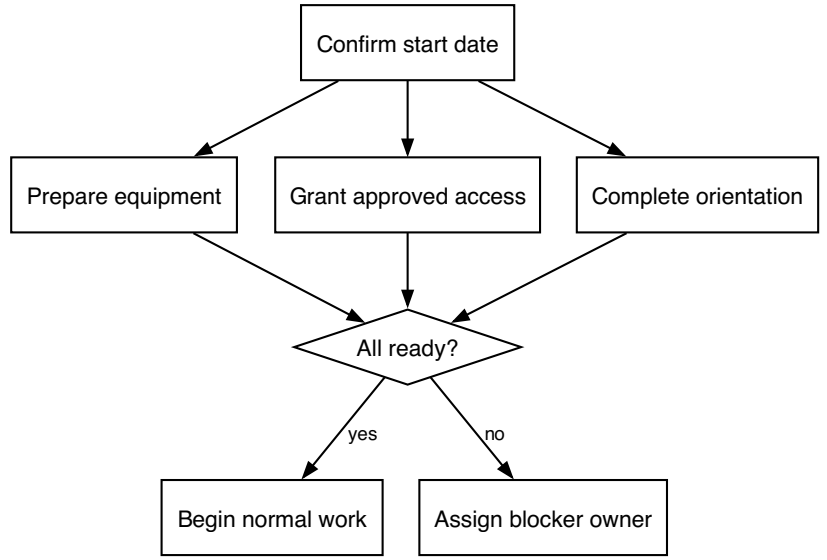
Equipment, access and introductory training can progress independently. The final gate requires all three to be complete. A blocked prerequisite routes to an owner rather than silently treating the new colleague as ready. AND is a stated convention, not an executable join.

Try changing: Split a large prerequisite into its own sub-process instead of enlarging every node.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  start [label="Confirm start date"];
  device [label="Prepare equipment"];
  access [label="Grant approved access"];
  intro [label="Complete orientation"];
  gate [shape=diamond, label="All ready?"];
  ready [label="Begin normal work"];
  blocked [label="Assign blocker owner"];
  start -> { device access intro };
  { device access intro } -> gate;
  gate -> ready [label="yes"];
  gate -> blocked [label="no"];
}
```

Actual output



GVZ-09-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-09-006

Include learning after restoration

Restoring service is not the last activity in this incident workflow. The closing review records follow-up work and feeds improvements back into preparation. The feedback edge is dotted and unconstrained so it remains visible without distorting the forward sequence.

Try changing: Use a separate detailed diagram for investigation paths that would overwhelm the overview.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  prepare [label="Prepare response"];
  detect [label="Detect and triage"];
  contain [label="Contain impact"];
  restore [label="Restore service"];
  review [label="Review and assign actions"];
  prepare -> detect -> contain -> restore -> review;
  review -> prepare [label="improve", style=dotted,
                     constraint=false];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-09-006 · Rendered by Graphviz

Graphviz · exit 0

Visual language and design

GVZ-10-001

Use a small semantic palette

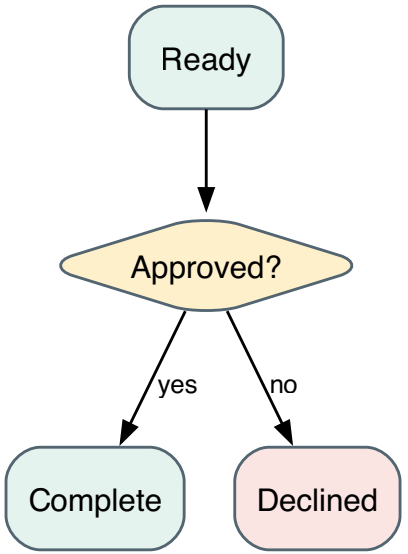
Color reinforces a meaning already carried by words and shapes. Green denotes completed work, amber denotes a decision, and red denotes rejection in this example. Filled nodes need fillcolor; color controls their outline. Keep the same mapping across the entire process family.

Try changing: Change the palette while preserving the labels and shape distinctions.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [style="rounded,filled", color="#526570"];
  ready [label="Ready", fillcolor="#e4f3ee"];
  gate [shape=diamond, label="Approved?",
        fillcolor="#fff0cc"];
  done [label="Complete", fillcolor="#e4f3ee"];
  stop [label="Declined", fillcolor="#fbe5e3"];
  ready -> gate;
  gate -> done [label="yes"];
  gate -> stop [label="no"];
}
```

Actual output



GVZ-10-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-10-002

Make the design work in monochrome

The same approval process remains understandable without colored fills. A heavier outline emphasizes the completed outcome, while the declined branch is dashed and labeled. Redundant visual cues help when a diagram is photocopied, printed in grayscale or read by someone who cannot distinguish its colors.

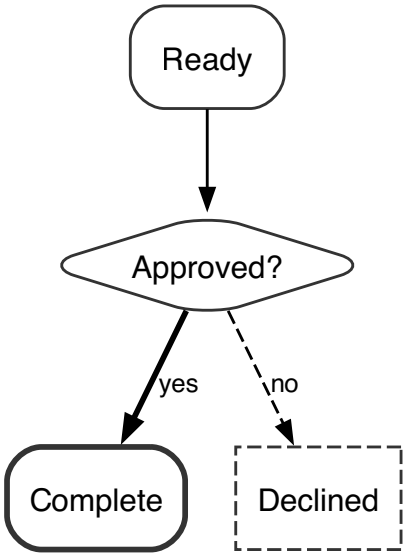
Try changing: Compare with the colored version to confirm that no meaning depends on hue.

Compare with GVZ-10-001 · Use a small semantic palette.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [style="rounded,filled", color="#333333",
        fillcolor="white"];
  ready [label="Ready"];
  gate [shape=diamond, label="Approved?"];
  done [label="Complete", penwidth=2];
  stop [label="Declined", style="dashed,filled"];
  ready -> gate;
  gate -> done [label="yes", penwidth=2];
  gate -> stop [label="no", style=dashed];
}
```

Actual output



GVZ-10-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-10-003

Emphasize the path you want readers to follow

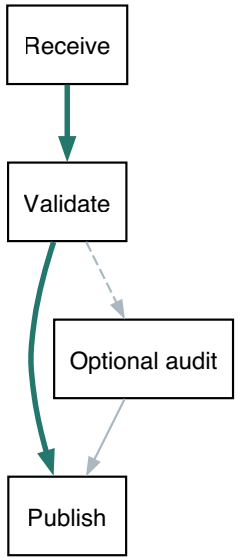
A strong line marks the primary route through the process, while a secondary review branch stays lighter. penwidth changes visual stroke thickness; weight affects dot layout and is a different control. This highlight is authored emphasis, not a calculated critical path or timing analysis.

Try changing: Highlight a user journey or exception path without changing the underlying connections.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  edge [color="#aab8c2"];
  receive [label="Receive"];
  validate [label="Validate"];
  publish [label="Publish"];
  audit [label="Optional audit"];
  receive -> validate -> publish
    [color="#24776d", penwidth=2.5];
  validate -> audit [style=dashed];
  audit -> publish;
}
```

Actual output



GVZ-10-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-10-004

Set typography at three levels

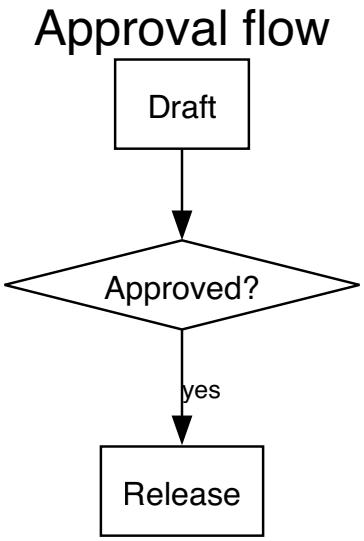
Graph, node and edge text have separate defaults. A graph title can be larger than activity names, and branch labels can be smaller. fontname requests an installed font; a fallback may change text metrics. Use short, consistent labels before trying to solve crowding by shrinking type.

Try changing: Increase only the graph title size and compare the text hierarchy.

DOT source

```
digraph process {
     graph [label="Approval flow", labelloc=t,
            fontname="Helvetica", fontsize=20];
     node [shape=box, fontname="Helvetica", fontsize=13];
     edge [fontname="Helvetica", fontsize=10];
     draft [label="Draft"];
     gate [shape=diamond, label="Approved?"];
     release [label="Release"];
     draft -> gate;
     gate -> release [label="yes"];
   }
```

Actual output



GVZ-10-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-10-005

Control padding and minimum node size

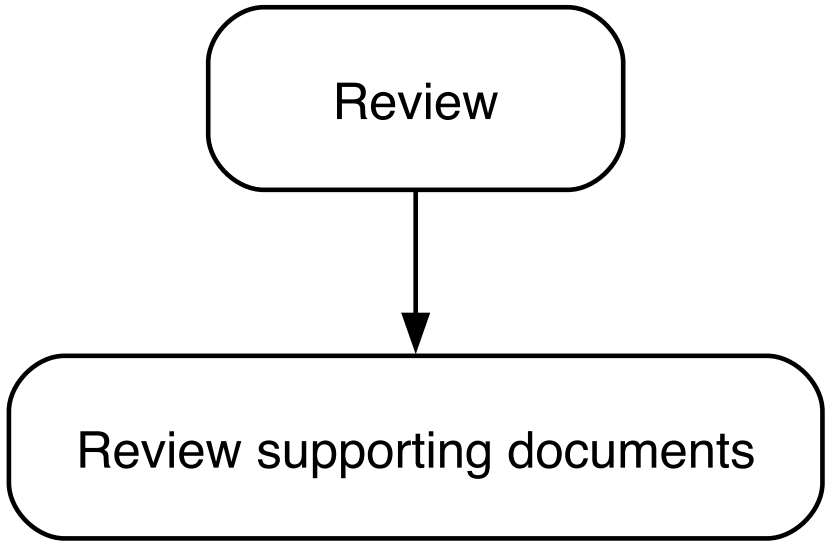
Node margin adds breathing room around text. Width and height are minimum dimensions when fixedsize=false, allowing long labels to enlarge the box. Avoid fixedsize=true for arbitrary text: it can produce labels that do not fit. Inch-based dimensions are independent of the font size in points.

Try changing: Reduce the margin before reducing the type size when a diagram needs to be more compact.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [style=rounded, margin="0.22,0.12",
        width=1.25, height=0.55, fixedsize=false];
  short [label="Review"];
  long [label="Review supporting documents"];
  short -> long;
}
```

Actual output



GVZ-10-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-10-006

Put the legend next to the process

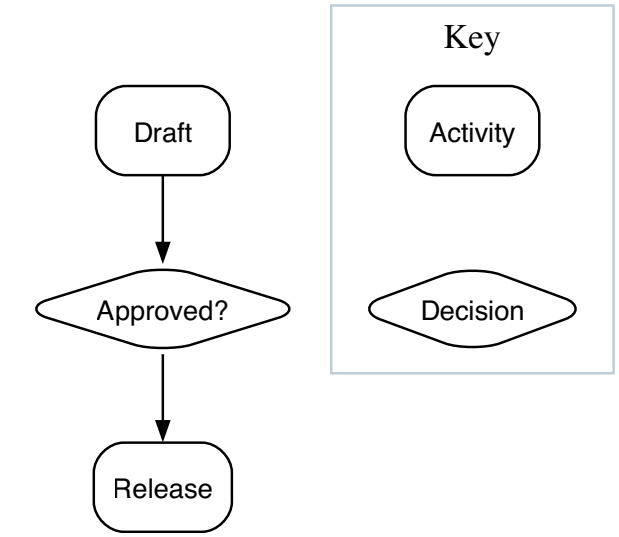
A legend defines the diagram's vocabulary where readers need it. The process uses the same rounded activity and diamond decision shapes shown in the key. Invisible edges arrange the legend only; the key is a disconnected component and does not participate in the workflow.

Try changing: Add only symbols actually used in the diagram; a large unused legend adds visual noise.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [style=rounded];
  draft [label="Draft"];
  approve [shape=diamond, label="Approved?"];
  release [label="Release"];
  draft -> approve -> release;
  subgraph cluster_key {
    label="Key"; color="#bdcdd9";
    action [label="Activity"];
    decision [shape=diamond, label="Decision"];
    action -> decision [style=invis];
  }
}
```

Actual output



GVZ-10-006 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Ranks, spacing and order

GVZ-11-001

Choose a reading direction

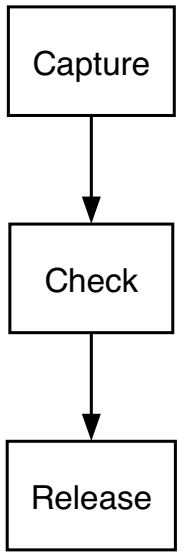
rankdir affects dot's main progression. TB reads top-to-bottom, LR reads left-to-right, and BT or RL reverse those directions. Use the available page shape and the audience's reading task to choose. A direction setting does not guarantee exact coordinates for individual nodes.

Try changing: Change only rankdir to LR, then RL, and compare the reading order.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  capture [label="Capture"];
  check [label="Check"];
  release [label="Release"];
  capture -> check -> release;
}
```

Actual output



GVZ-11-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-11-002

Separate rank spacing from node spacing

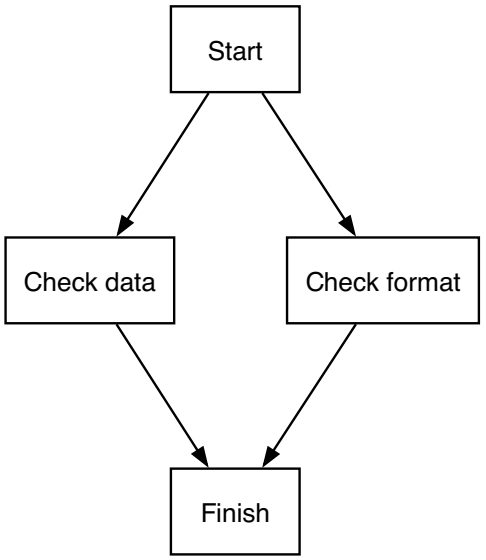
ranksep controls the gap between successive ranks; nodesep controls separation between adjacent nodes on the same rank in dot. Both use inches. The two branches make the difference visible: one setting opens the diagram along the main flow, the other spreads the alternatives apart.

Try changing: Halve nodesep while leaving ranksep unchanged, then reverse the experiment.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  ranksep=0.85;
  nodesep=0.65;
  start [label="Start"];
  left [label="Check data"];
  right [label="Check format"];
  end [label="Finish"];
  start -> { left right } -> end;
  { rank=same; left; right; }
}
```

Actual output



GVZ-11-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-11-003

Reserve space for a longer dependency

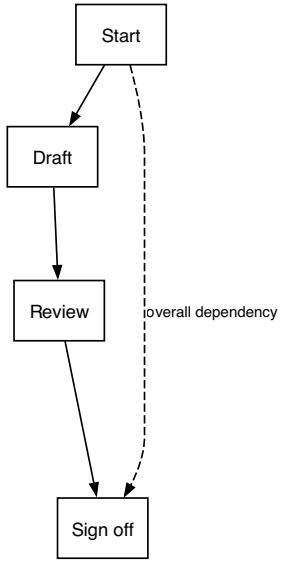
minlen sets a minimum rank difference for one edge. It counts ranks rather than inches, unlike ranksep. Here the direct signoff dependency spans at least five ranks, giving the intermediate work room to sit alongside it. The setting is a layout constraint, not an elapsed-time annotation.

Try changing: Compare minlen=1 and minlen=5 while keeping the same process edges.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  start [label="Start"];
  draft [label="Draft"];
  review [label="Review"];
  signoff [label="Sign off"];
  start -> draft -> review -> signoff;
  start -> signoff [minlen=5, style=dashed,
                    label="overall dependency"];
}
```

Actual output



GVZ-11-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-11-004

Keep branch order stable

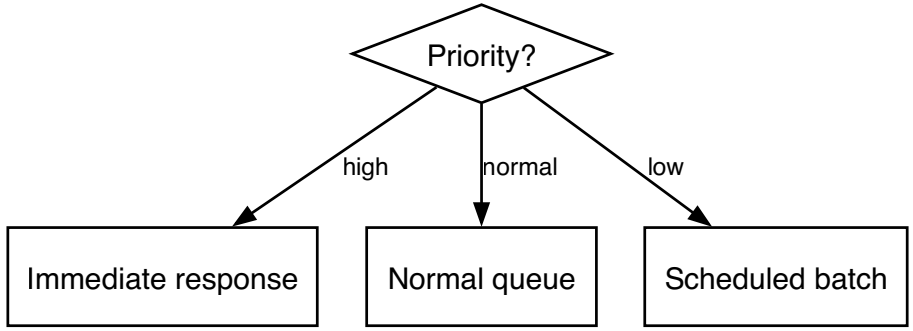
ordering=out asks dot to place a node's outgoing edges left-to-right in their source order. This is useful when branches should read in a deliberate order. It controls edge ordering, not arbitrary global coordinates; other constraints still affect the rest of the drawing.

Try changing: Swap two outgoing edge statements and inspect which branches change position.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  ordering=out;
  choose [shape=diamond, label="Priority?"];
  urgent [label="Immediate response"];
  normal [label="Normal queue"];
  low [label="Scheduled batch"];
  choose -> urgent [label="high"];
  choose -> normal [label="normal"];
  choose -> low [label="low"];
}
```

Actual output



GVZ-11-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-11-005

Use invisible edges as layout scaffolding

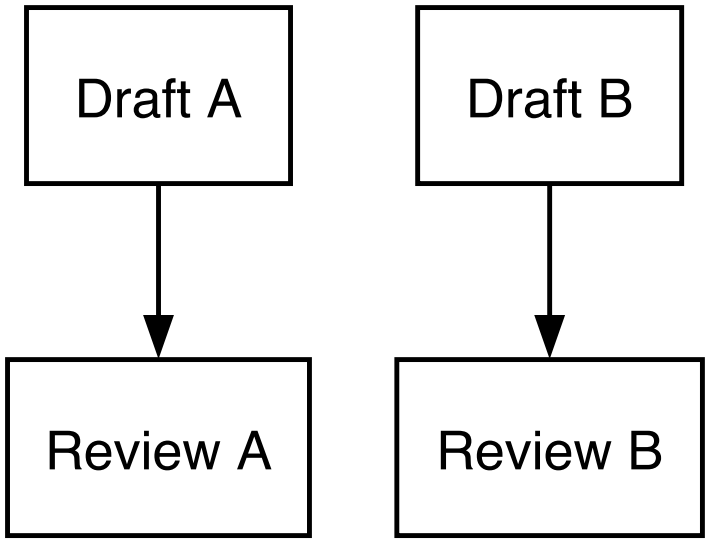
The visible arrows represent two streams. Matching group values encourage straight connections within each stream, while invisible same-rank edges establish a left-to-right ordering. These invisible edges are layout aids only; the accompanying explanation prevents them from being mistaken for hidden process requirements.

Try changing: Remove the group values separately from the invisible edges to isolate each effect.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  a1 [label="Draft A", group=a];
  a2 [label="Review A", group=a];
  b1 [label="Draft B", group=b];
  b2 [label="Review B", group=b];
  { rank=same; a1; b1; }
  { rank=same; a2; b2; }
  a1 -> a2;
  b1 -> b2;
  a1 -> b1 [style=invis];
  a2 -> b2 [style=invis];
}
```

Actual output



Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-11-006

Anchor the entry and exit ranks

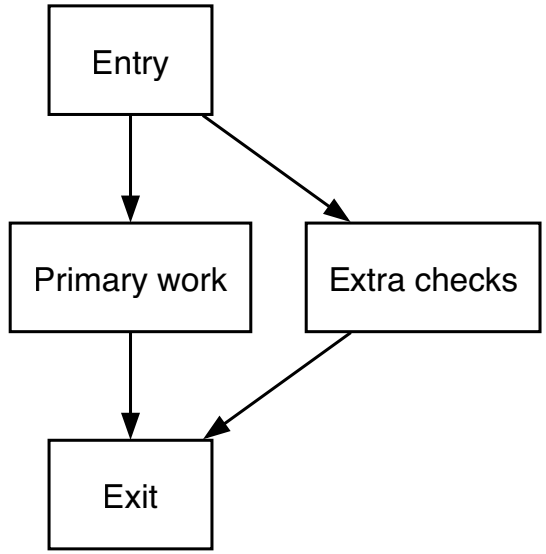
rank=source and rank=sink reserve the first and last ranks for the chosen nodes. This helps when several internal paths have different lengths. The heavier main edge encourages a shorter, straighter route in dot; weight does not make the edge thicker or change process priority.

Try changing: Set both paths to the same weight to see whether the preferred alignment changes.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  entry [label="Entry"];
  primary [label="Primary work"];
  optional [label="Extra checks"];
  exit [label="Exit"];
  { rank=source; entry; }
  { rank=sink; exit; }
  entry -> primary -> exit [weight=8];
  entry -> optional -> exit;
}
```

Actual output



GVZ-11-006 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Edge routing and connections

GVZ-12-001

Route smooth splines

splines=spline lets dot route curved connections around nodes. The direct bypass from Start to Finish bends around the intermediate tasks, making the routing easy to compare with the next two examples. Routing changes the drawn path of an edge while leaving its endpoints and process relationship unchanged.

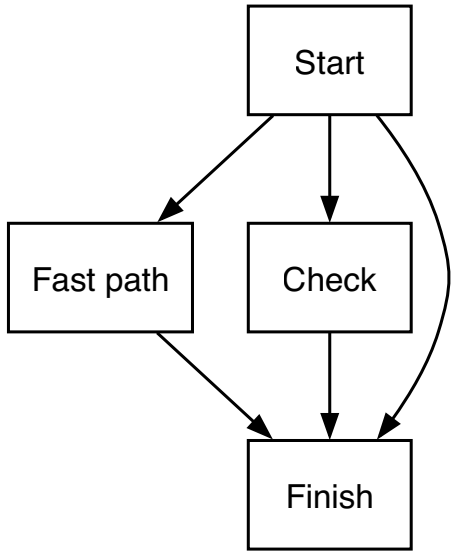
Try changing: Compare the next two examples before choosing a routing style for a whole diagram family.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  splines=spline;

  start [label="Start"];
  fast [label="Fast path"];
  check [label="Check"];
  end [label="Finish"];
  start -> fast -> end;
  start -> check -> end;
  start -> end;
  { rank=same; fast; check; }
}
```

Actual output



GVZ-12-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-12-002

Use segmented polylines

Only the routing setting changes from GVZ-12-001. splines=polyline uses piecewise straight segments, which can make turns more explicit. Check the result at its final size: more corners do not necessarily make a dense process easier to follow.

Try changing: Look for crossings and sharp turns rather than judging the style only on a small example.

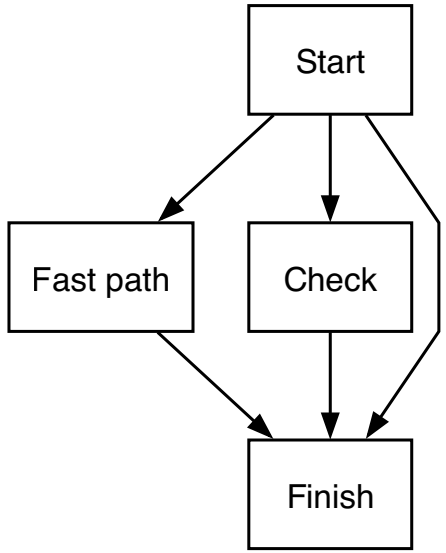
Compare with GVZ-12-001 · Route smooth splines.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  splines=polyline;

  start [label="Start"];
  fast [label="Fast path"];
  check [label="Check"];
  end [label="Finish"];
  start -> fast -> end;
  start -> check -> end;
  start -> end;
  { rank=same; fast; check; }
}
```

Actual output



GVZ-12-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-12-003

Use orthogonal routing carefully

splines=ortho requests axis-aligned segments. Graphviz documents limitations with ports and ordinary edge labels in this mode, so this example uses neither. For a labeled decision diagram, prefer splines or polylines, or test external xlabels carefully after layout.

Try changing: Compare the corners with the spline baseline; avoid combining ortho with explicit ports.

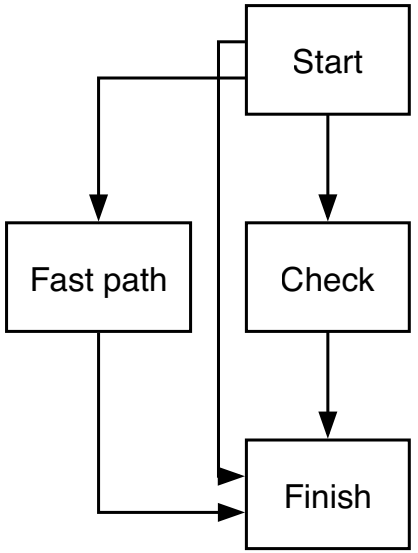
Compare with GVZ-12-001 · Route smooth splines.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  splines=ortho;

  start [label="Start"];
  fast [label="Fast path"];
  check [label="Check"];
  end [label="Finish"];
  start -> fast -> end;
  start -> check -> end;
  start -> end;
  { rank=same; fast; check; }
}
```

Actual output



GVZ-12-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-12-004

Choose where edges meet a node

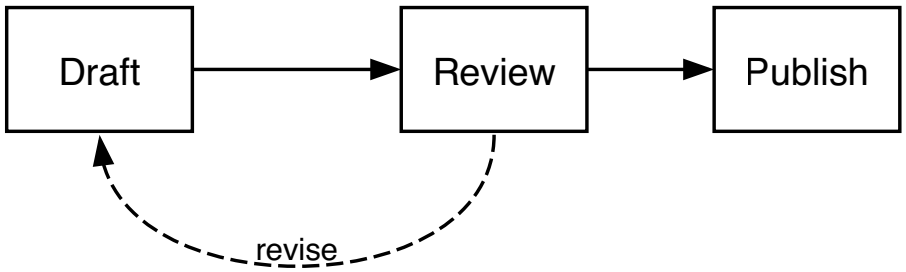
Compass ports identify sides of a node: n, s, e and w mean north, south, east and west. With ordinary spline routing, the two paths can attach to different sides of Review. A port guides the endpoint rather than prescribing the entire path.

Try changing: Move the return edge to the north ports and compare the loop placement.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  splines=spline;
  draft [label="Draft"];
  review [label="Review"];
  publish [label="Publish"];
  draft:e -> review:w;
  review:e -> publish:w;
  review:s -> draft:s [label="revise",
                       constraint=false, style=dashed];
}
```

Actual output



GVZ-12-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-12-005

Keep feedback from pulling the main flow

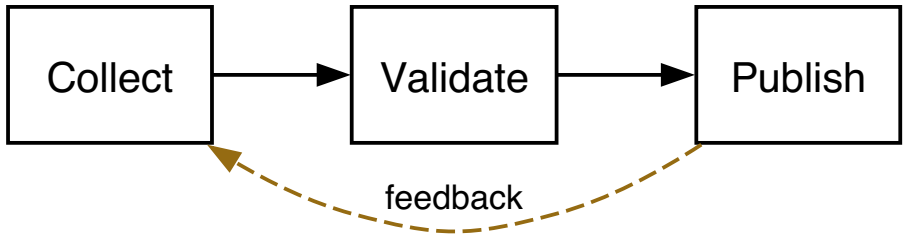
The return edge documents rework, while constraint=false removes that edge from rank assignment in dot. It still exists and is still drawn. A larger weight on the forward path encourages a straighter main sequence, independently of the feedback edge's dashed appearance.

Try changing: Remove constraint=false and inspect the layout before deciding whether it is needed.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  collect [label="Collect"];
  validate [label="Validate"];
  publish [label="Publish"];
  collect -> validate -> publish [weight=6];
  publish -> collect [label="feedback", constraint=false,
                      style=dashed, color="#956b12"];
}
```

Actual output



GVZ-12-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-12-006

Connect cluster boundaries

compound=true enables edges clipped at cluster boundaries. The edge still connects real member nodes, but ltail and lhead name the enclosing source and destination clusters. This is useful for an overview handoff between sub-processes, without pretending the cluster itself is an ordinary node.

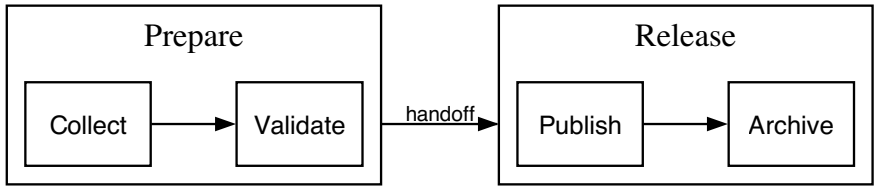
Try changing: Remove lhead and ltail to compare node-to-node and boundary-to-boundary connections.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  compound=true;
  subgraph cluster_prepare {
    label="Prepare";
    collect [label="Collect"];
    validate [label="Validate"];
    collect -> validate;
  }
  subgraph cluster_release {
    label="Release";
    publish [label="Publish"];
    archive [label="Archive"];
    publish -> archive;
  }
  validate -> publish [ltail=cluster_prepare,
                       lhead=cluster_release,
                       label="handoff"];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-12-006 · Rendered by Graphviz

Graphviz · exit 0

Rich labels and annotations

GVZ-13-001

Build an activity card with a subtitle

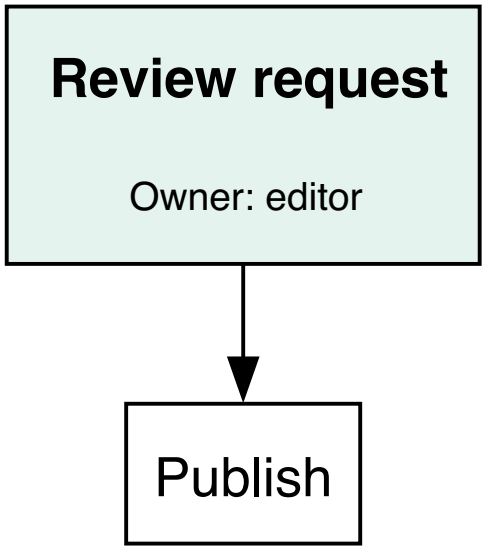
An HTML-like label can establish a title and smaller supporting line inside one node. shape=plain lets the table determine the boundary. CELLBORDER=0 removes interior rules, CELLPADDING adds inset space, and BGCOLOR provides a restrained background. This is Graphviz's label markup, not arbitrary browser HTML.

Try changing: Change the subtitle to an input, output or owner; avoid turning every card into a paragraph.

DOT source

```
digraph process {
     node [shape=plain, fontname="Helvetica"];
     review [label=<
       <TABLE BORDER="1" CELLBORDER="0" CELLPADDING="8"
              BGCOLOR="#e4f3ee">
         <TR><TD><B>Review request</B></TD></TR>
         <TR><TD><FONT POINT-SIZE="10">Owner: editor</FONT></TD></TR>
       </TABLE>
     >];
     publish [shape=box, label="Publish"];
     review -> publish;
   }
```

Actual output



GVZ-13-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-13-002

Give inputs and outputs named ports

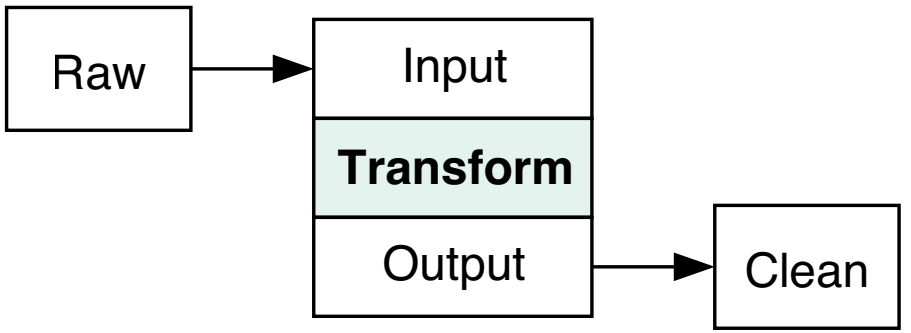
A table can name distinct attachment points with PORT. The source connects to the input cell and the output cell connects to the destination. This works with normal spline routing and is useful for processes with several artifacts or channels. The port name is part of the connection syntax, not visible prose.

Try changing: Add a separate error port when a task produces both a normal result and a diagnostic artifact.

DOT source

```
digraph process {
     rankdir=LR;
     node [shape=plain, fontname="Helvetica"];
     task [label=<
       <TABLE BORDER="0" CELLBORDER="1" CELLSPACING="0"
              CELLPADDING="6">
         <TR><TD PORT="input">Input</TD></TR>
         <TR><TD BGCOLOR="#e4f3ee"><B>Transform</B></TD></TR>
         <TR><TD PORT="output">Output</TD></TR>
       </TABLE>
     >];
     raw [shape=box, label="Raw"];
     clean [shape=box, label="Clean"];
     raw:e -> task:input:w;
     task:output:e -> clean:w;
   }
```

Actual output



GVZ-13-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-13-003

Align multiline text deliberately

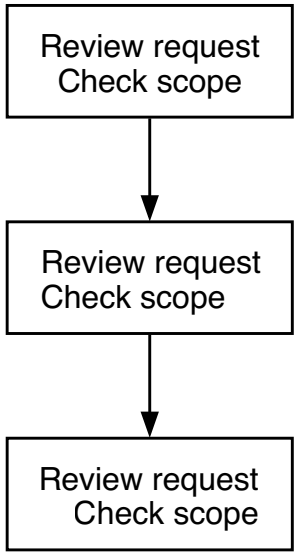
Quoted labels use backslash-n for centered line breaks, backslash-l for left-aligned lines, and backslash-r for right-aligned lines. The escape ends the line it aligns, including the final line. These are DOT label escapes; they are not the same as inserting an actual newline into the source file.

Try changing: Use left alignment for short lists and centered labels for compact activity names.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [margin="0.16,0.10"];
  centered [label="Review request\nCheck scope\n"];
  left [label="Review request\lCheck scope\l"];
  right [label="Review request\rCheck scope\r"];
  centered -> left -> right;
}
```

Actual output



GVZ-13-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-13-004

Label the relationship and its endpoints

label names the relationship, while taillabel and headlabel annotate its ends. labelfontsize applies to those endpoint labels; fontsize applies to the ordinary edge label. labeldistance adjusts their offset and labelangle their direction. A wider ranksep leaves room for all three labels. Inspect the rendered result because crowded endpoint labels can collide.

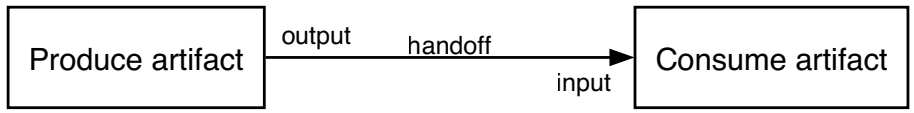
Try changing: Compare endpoint annotations with a single short edge label before adding more text.

DOT source

```
digraph process {
  rankdir=LR;
  ranksep=1.4;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  produce [label="Produce artifact"];
  consume [label="Consume artifact"];
  produce -> consume [label="handoff", taillabel="output",
                      headlabel="input", labelfontsize=9,
                      labeldistance=2.0, labelangle=25];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-13-004 · Rendered by Graphviz

Graphviz · exit 0

GVZ-13-005

Attach a note without changing the process

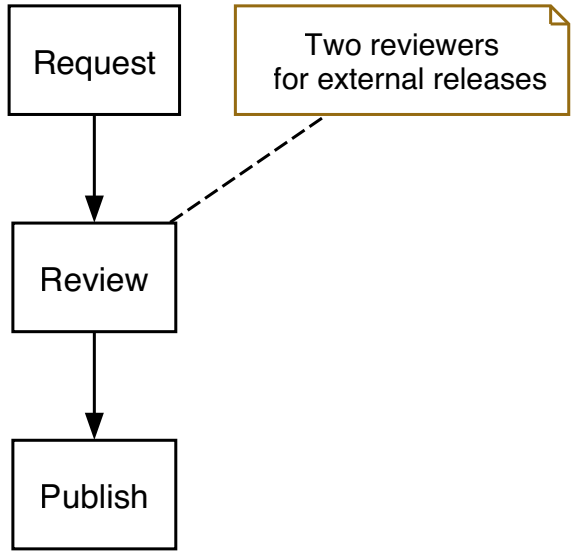
A note explains a step without becoming another activity. shape=note provides a paper-like symbol; dir=none removes the arrowhead and constraint=false keeps the annotation out of rank assignment. The dashed connector communicates association, so it should not be interpreted as a sequence transition.

Try changing: Move detailed policy text to a separate note instead of enlarging the activity label.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  request [label="Request"];
     review [label="Review"];
     publish [label="Publish"];
     note [shape=note, label="Two reviewers
  for external releases",
           fontsize=10, color="#956b12"];
     request -> review -> publish;
     review -> note [style=dashed, dir=none, constraint=false];
}
```

Actual output



GVZ-13-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-13-006

Add a title and a scope statement

A graph label provides context that travels with the downloaded SVG. labelloc=t puts it above the drawing; labeljust=l requests left justification. An HTML-like label makes the process name prominent and the scope line smaller. This is a useful place to state the diagram's boundary or revision.

Try changing: Replace the scope statement with a useful revision or ownership note, keeping it short.

DOT source

```
digraph process {
     graph [labelloc=t, labeljust=l, fontname="Helvetica",
            label=<<B>Content approval</B><BR/>
              <FONT POINT-SIZE="10">Scope: draft to release</FONT>>];
     rankdir=LR;
     node [shape=box, fontname="Helvetica", fontsize=11];
     draft [label="Draft"];
     review [label="Review"];
     release [label="Release"];
     draft -> review -> release;
   }
```

Actual output



GVZ-13-006 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Composing larger process diagrams

GVZ-14-001

Nest phases inside an overall process

Nested clusters show a hierarchy: the overall process contains preparation and delivery phases. Distinct cluster labels establish scope while the arrows still connect activities. Avoid deep nesting; each extra boundary consumes space and can make crossings harder to read.

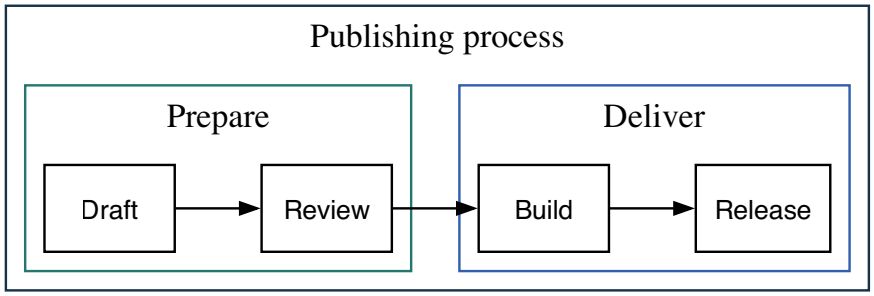
Try changing: Split a third level into its own diagram when the nested boundaries dominate the content.

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  subgraph cluster_process {
    label="Publishing process"; color="#173147";
    subgraph cluster_prepare {
      label="Prepare"; color="#24776d";
      draft [label="Draft"];
      review [label="Review"];
      draft -> review;
    }
    subgraph cluster_deliver {
      label="Deliver"; color="#315eaf";
      build [label="Build"];
      release [label="Release"];
      build -> release;
    }
    review -> build;
  }
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-14-001 · Rendered by Graphviz

Graphviz · exit 0

GVZ-14-002

Align three stages across two owners

This extends the two-lane pattern to a shared three-stage grid. newrank=true permits rank constraints across clusters. Invisible edges align positions within each lane, and same-rank groups align the stages. Only the solid arrows are process steps; the dashed arrow is a review return.

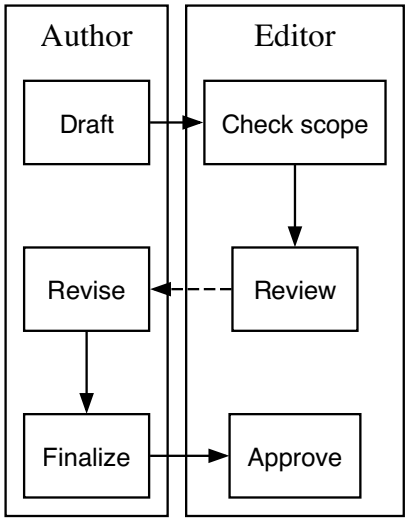
Try changing: Keep scaffold edges visibly separate in the source so they are not confused with business logic.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  newrank=true;
  subgraph cluster_author {
    label="Author";
    a1 [label="Draft", group=a];
    a2 [label="Revise", group=a];
    a3 [label="Finalize", group=a];
    a1 -> a2 [style=invis];
  }
  subgraph cluster_editor {
    label="Editor";
    b1 [label="Check scope", group=b];
    b2 [label="Review", group=b];
    b3 [label="Approve", group=b];
    b1 -> b2;
    b2 -> b3 [style=invis];
  }
  { rank=same; a1; b1; }
  { rank=same; a2; b2; }
  { rank=same; a3; b3; }
  a1 -> b1 [constraint=false];
  b2 -> a2 [constraint=false, style=dashed];
  a2 -> a3;
  a3 -> b3 [constraint=false];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-14-002 · Rendered by Graphviz

Graphviz · exit 0

GVZ-14-003

Pack independent workflows as small multiples

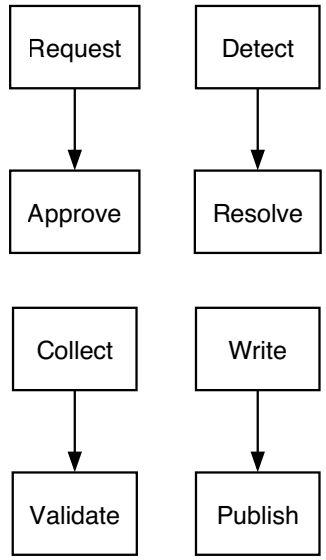
Disconnected components can be laid out separately and packed together. pack specifies padding in points, while packmode=array2 arranges components in two columns. These panels are independent workflows: packing changes their placement but does not add relationships between them.

Try changing: Change array2 to array1 to compare a dashboard-like arrangement with a vertical reference.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  pack=24;
  packmode="array2";
  a1 [label="Write"];
  a2 [label="Publish"];
  b1 [label="Request"];
  b2 [label="Approve"];
  c1 [label="Collect"];
  c2 [label="Validate"];
  d1 [label="Detect"];
  d2 [label="Resolve"];
  a1 -> a2;
  b1 -> b2;
  c1 -> c2;
  d1 -> d2;
}
```

Actual output



GVZ-14-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-14-004

Show detail through a named sub-process

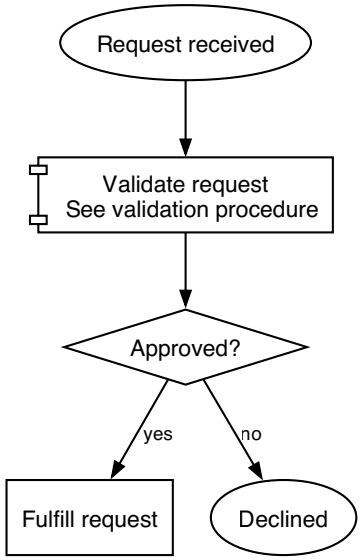
A compact overview uses a named sub-process instead of exposing every internal task. The component-shaped symbol is a visual convention for a reusable procedure. This is an abstraction decision: keep the entry condition, outcome and detail reference in the documentation so the overview remains traceable.

Try changing: Replace a crowded cluster with a named sub-process and link its detailed definition separately.

DOT source

```
digraph process {
  rankdir=TB;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  start [shape=ellipse, label="Request received"];
     validate [shape=component, label="Validate request
  See validation procedure"];
     approve [shape=diamond, label="Approved?"];
     fulfill [label="Fulfill request"];
     reject [shape=ellipse, label="Declined"];
     start -> validate -> approve;
     approve -> fulfill [label="yes"];
     approve -> reject [label="no"];
}
```

Actual output



GVZ-14-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-14-005

Compose a finished approval diagram

This complete recipe combines a restrained palette, readable labels, a decision gate and an explicit revision loop. The main path uses green emphasis; the return path uses amber and dashes. A graph title gives the exported image context. Keep each cue consistent with the earlier design examples.

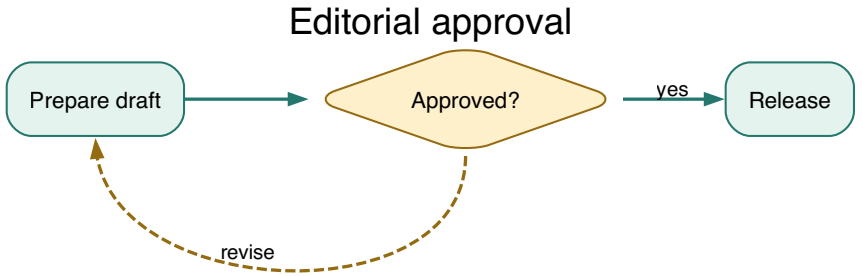
Try changing: Adapt the text to a real approval process before adding further decoration.

DOT source

```
digraph approval {
     graph [rankdir=LR, pad=0.25, nodesep=0.4,
            label="Editorial approval", labelloc=t,
            fontname="Helvetica", fontsize=18];
     node [shape=box, style="rounded,filled", margin="0.16,0.10",
           fontname="Helvetica", fontsize=11,
           color="#24776d", fillcolor="#e4f3ee"];
     edge [fontname="Helvetica", fontsize=10,
           color="#24776d", penwidth=1.6, arrowsize=0.7];
     draft [label="Prepare draft"];
     review [shape=diamond, label="Approved?",
             fillcolor="#fff0cc", color="#956b12"];
     release [label="Release"];
     draft -> review;
     review -> release [label="yes"];
     review:s -> draft:s [label="revise", constraint=false,
                          style=dashed, color="#956b12"];
   }
```

Download: DOT source · SVG diagram

Actual output



GVZ-14-005 · Rendered by Graphviz

Graphviz · exit 0

GVZ-14-006

Compose a process with artifacts and owners

The final recipe combines role boundaries, an artifact, a decision and rework. The source file and published output are explicitly named; the author and editor own different activities. This is an overview of responsibility and flow, not a formal BPMN model or an executable workflow definition.

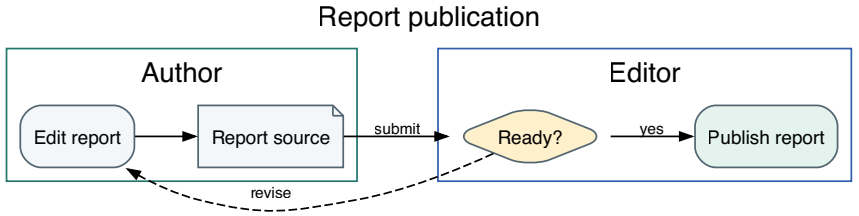
Try changing: Check every boundary crossing: who sends what, who receives it, and what returns on failure?

DOT source

```
digraph process {
  rankdir=LR;
  node [shape=box, fontname="Helvetica", fontsize=11];
  edge [fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [pad=0.2, label="Report publication", labelloc=t,
         fontname="Helvetica", fontsize=16];
  node [style="rounded,filled", fillcolor="#f3f7fa",
        color="#526570"];
  subgraph cluster_author {
    label="Author"; color="#24776d";
    edit [label="Edit report"];
    source [shape=note, label="Report source"];
    edit -> source;
  }
  subgraph cluster_editor {
    label="Editor"; color="#315eaf";
    review [shape=diamond, label="Ready?", fillcolor="#fff0cc"];
    publish [label="Publish report", fillcolor="#e4f3ee"];
    review -> publish [label="yes"];
  }
  source -> review [label="submit"];
  review -> edit [label="revise", style=dashed, constraint=false];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-14-006 · Rendered by Graphviz

Graphviz · exit 0

Architecture and dependencies

GVZ-15-001

Map a service architecture

Boxes represent deployed components, cylinders represent storage, and arrows show which component calls or reads another. The two clusters separate request handling from stored data. This is a structural view: arrow direction does not prescribe the order of a single request. Shared storage becomes visible because several services point to the same node.

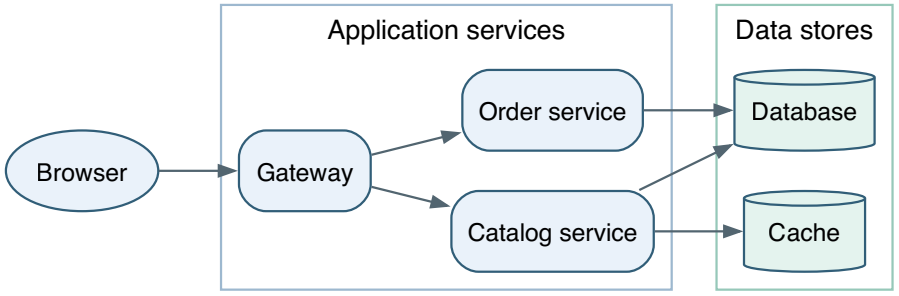
Try changing: Add a reporting service and connect only the stores it actually reads.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  browser [label="Browser", shape=oval];
  subgraph cluster_services {
    label="Application services"; color="#9db8cf";
    gateway [label="Gateway"];
    catalog [label="Catalog service"];
    orders [label="Order service"];
    gateway -> {catalog orders};
  }
  subgraph cluster_data {
    label="Data stores"; color="#96c8b9";
    node [shape=cylinder, fillcolor="#e4f3ee"];
    database [label="Database"];
    cache [label="Cache"];
  }
  browser -> gateway;
  catalog -> {database cache};
  orders -> database;
}
```

Download: DOT source · SVG diagram

Actual output



Graphviz · exit 0

GVZ-15-002

Separate build actions from artifacts

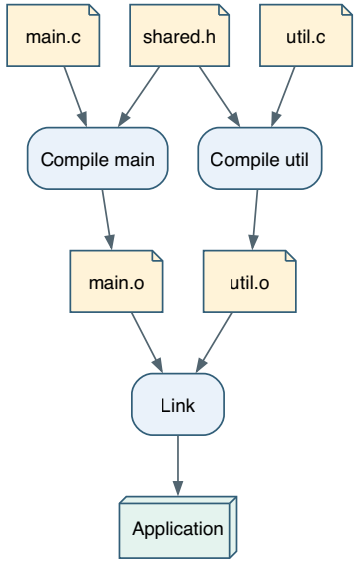
A note is a file; a rounded box is a build action. Two compilation branches join at the linker, making the shared header and the required object files visible. The graph describes prerequisites, so it can explain which branch needs rebuilding after a source edit. It does not run a compiler or calculate file freshness.

Try changing: Trace the affected path when util.c changes, then compare it with a shared-header change.

DOT source

```
digraph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=note, style=filled, fillcolor="#fff3d8"];
  main [label="main.c"]; util [label="util.c"];
  header [label="shared.h"];
  main_o [label="main.o"]; util_o [label="util.o"];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa"];
  compile_main [label="Compile main"];
  compile_util [label="Compile util"];
  link [label="Link"];
  app [shape=box3d, label="Application", fillcolor="#e4f3ee"];
  main -> compile_main; header -> compile_main;
  util -> compile_util; header -> compile_util;
  compile_main -> main_o; compile_util -> util_o;
  {main_o util_o} -> link;
  link -> app;
}
```

Actual output



GVZ-15-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-15-003

Trace the lineage of a derived dataset

This view follows data products rather than activities. Two sources contribute to an enriched table, which feeds a summary and two consumers. Edge labels name transformations; cylinders name persisted datasets. Keeping an intermediate dataset as a node makes reuse and downstream impact easier to inspect than a single long pipeline.

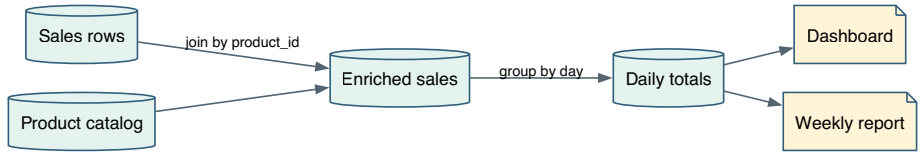
Try changing: Add a second consumer of the enriched table without duplicating the dataset node.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=cylinder, style=filled, fillcolor="#e4f3ee"];
  sales [label="Sales rows"];
  products [label="Product catalog"];
  enriched [label="Enriched sales"];
  totals [label="Daily totals"];
  node [shape=note, fillcolor="#fff3d8"];
  dashboard [label="Dashboard"];
  report [label="Weekly report"];
  {rank=same; sales; products;}
  sales -> enriched [label="join by product_id"];
  products -> enriched;
  enriched -> totals [label="group by day"];
  totals -> {dashboard report};
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-15-003 · Rendered by Graphviz

Graphviz · exit 0

GVZ-15-004

Show event distribution to independent subscribers

Two publishers emit events to a topic. Three subscribers receive those events independently; the drawing does not imply a work queue where only one consumer receives each message. The hexagon emphasizes the shared distribution point. Delivery guarantees and ordering are application choices and are not determined by this fan-out shape.

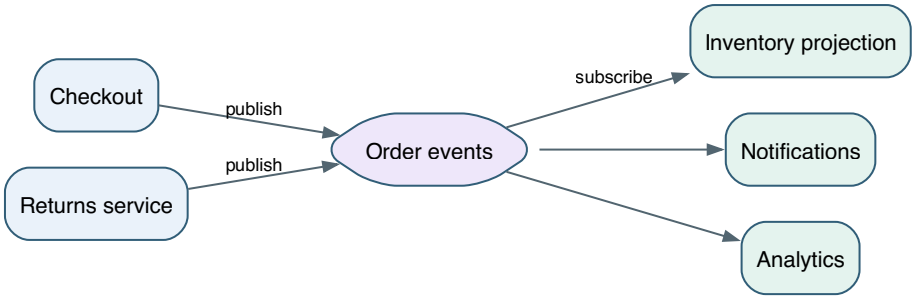
Try changing: Add an audit subscriber and describe whether it needs every event or only a filtered subset.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  checkout [label="Checkout"];
  returns [label="Returns service"];
  topic [shape=hexagon, label="Order events", fillcolor="#eee7fa"];
  node [fillcolor="#e4f3ee"];
  inventory [label="Inventory projection"];
  notifications [label="Notifications"];
  analytics [label="Analytics"];
  {rank=same; checkout; returns;}
  {rank=same; inventory; notifications; analytics;}
  checkout -> topic [label="publish"];
  returns -> topic [label="publish"];
  topic -> inventory [label="subscribe"];
  topic -> notifications;
  topic -> analytics;
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-15-004 · Rendered by Graphviz

Graphviz · exit 0

GVZ-15-005

Distinguish replicas from independent services

The two application nodes run the same role in different zones. Both write through the primary database; the dashed edge identifies replication to the standby. Cluster boundaries communicate placement, while labels communicate operational roles. A second database in the picture does not by itself imply automatic failover or consistent reads.

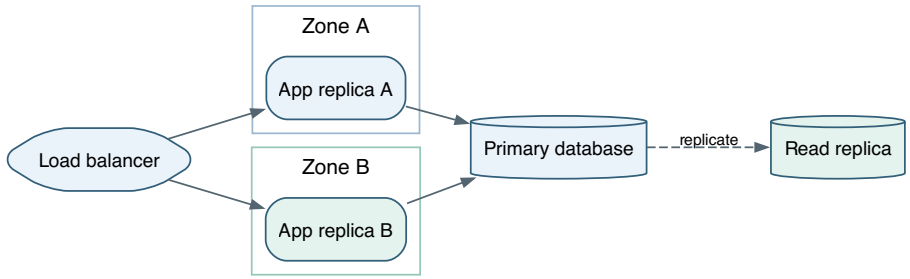
Try changing: Add a read-only consumer of the replica and label the freshness assumption.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  lb [label="Load balancer", shape=hexagon];
  subgraph cluster_a {
    label="Zone A"; color="#9db8cf";
    app_a [label="App replica A"];
  }
  subgraph cluster_b {
    label="Zone B"; color="#96c8b9";
    app_b [label="App replica B", fillcolor="#e4f3ee"];
  }
  primary [shape=cylinder, label="Primary database"];
  replica [shape=cylinder, label="Read replica", fillcolor="#e4f3ee"];
  lb -> {app_a app_b};
  {app_a app_b} -> primary;
  primary -> replica [label="replicate", style=dashed];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-15-005 · Rendered by Graphviz

Graphviz · exit 0

GVZ-15-006

Draw dependencies around an interface

Component-shaped nodes describe code boundaries. The application depends on an interface; an adapter implements that interface and accesses storage. The hollow arrow is an explicitly chosen implementation convention. These arrows describe source dependencies, not the direction in which runtime data necessarily travels.

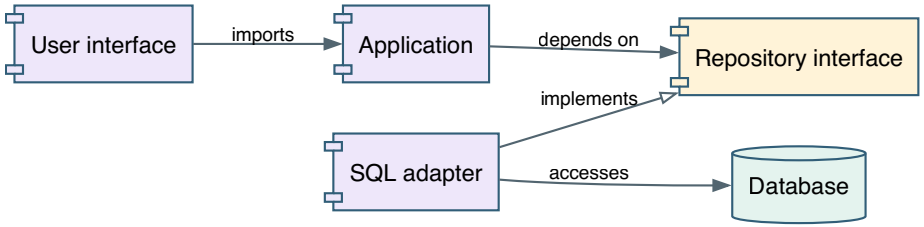
Try changing: Add an in-memory adapter that implements the same interface.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=component, style=filled, fillcolor="#eee7fa"];
  ui [label="User interface"];
  app [label="Application"];
  port [label="Repository interface", fillcolor="#fff3d8"];
  adapter [label="SQL adapter"];
  db [shape=cylinder, label="Database", fillcolor="#e4f3ee"];
  ui -> app [label="imports"];
  app -> port [label="depends on"];
  adapter -> port [arrowhead=empty, label="implements"];
  adapter -> db [label="accesses"];
  {rank=same; app; adapter;}
}
```

Download: DOT source · SVG diagram

Actual output



Graphviz · exit 0

Trees and relationship maps

GVZ-16-001

Draw an organization tree

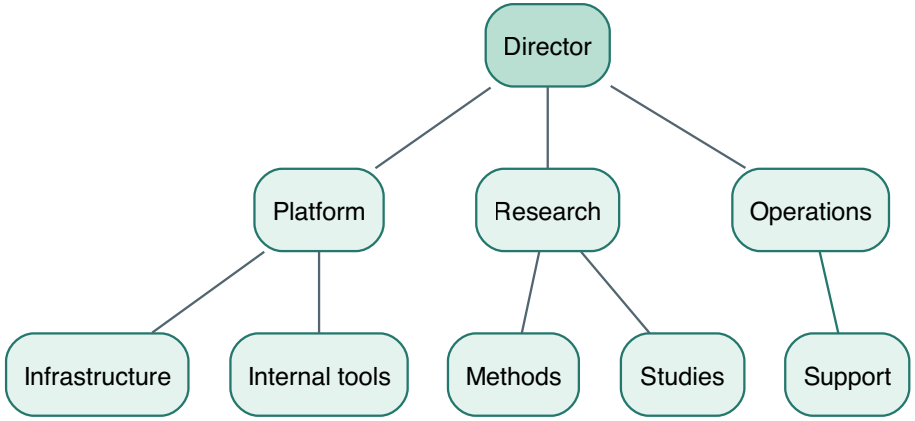
Each edge means reports to the node above. A tree layout gives reporting depth a consistent vertical direction, while same-rank siblings remain visually comparable. This small structure has one parent per team. A matrix organization needs a separately labeled relationship rather than an unexplained second reporting arrow.

Try changing: Add a team under Platform and keep the meaning of every edge the same.

DOT source

```
digraph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [fillcolor="#e4f3ee", color="#24776d"];
  director [label="Director", fillcolor="#b8dfd1"];
  platform [label="Platform"];
  research [label="Research"];
  operations [label="Operations"];
  infra [label="Infrastructure"]; tools [label="Internal tools"];
  methods [label="Methods"]; studies [label="Studies"];
  edge [arrowhead=none];
  director -> {platform research operations};
  platform -> {infra tools};
  research -> {methods studies};
  operations -> support [label="", color="#24776d"];
  support [label="Support"];
}
```

Actual output



Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-16-002

Build a taxonomy with uneven branches

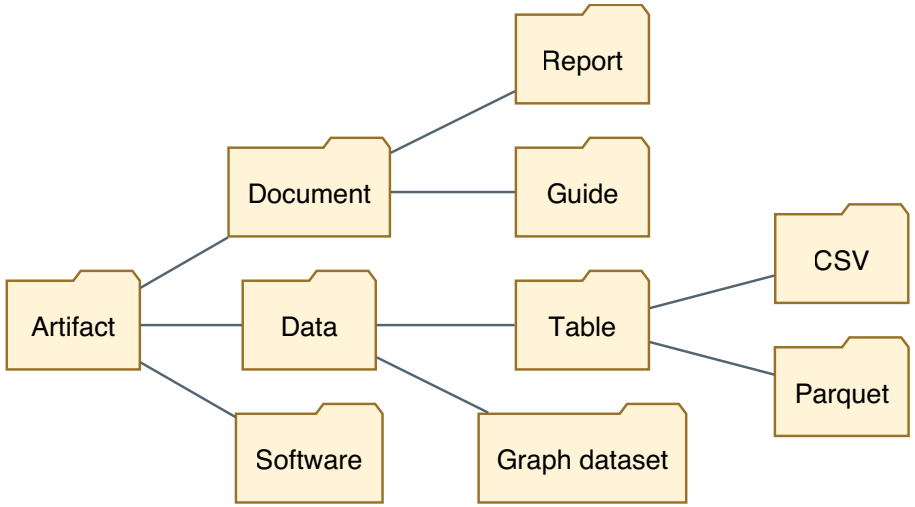
Folder-shaped nodes and arrowless connectors suggest classification, not execution. Some categories stop early while others contain another level; there is no need to invent filler nodes to make the tree balanced. The left-to-right direction leaves more horizontal room for category labels.

Try changing: Introduce a new format under Data and decide whether it belongs in one branch or requires a cross-classification.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=folder, style=filled, fillcolor="#fff3d8", color="#997128"];
  artifact [label="Artifact"];
  document [label="Document"]; data [label="Data"]; software [label="Software"];
  report [label="Report"]; guide [label="Guide"];
  table [label="Table"]; graph_data [label="Graph dataset"];
  csv [label="CSV"]; parquet [label="Parquet"];
  edge [arrowhead=none];
  artifact -> {document data software};
  document -> {report guide};
  data -> {table graph_data};
  table -> {csv parquet};
}
```

Actual output



GVZ-16-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-16-003

Visualize an expression as a syntax tree

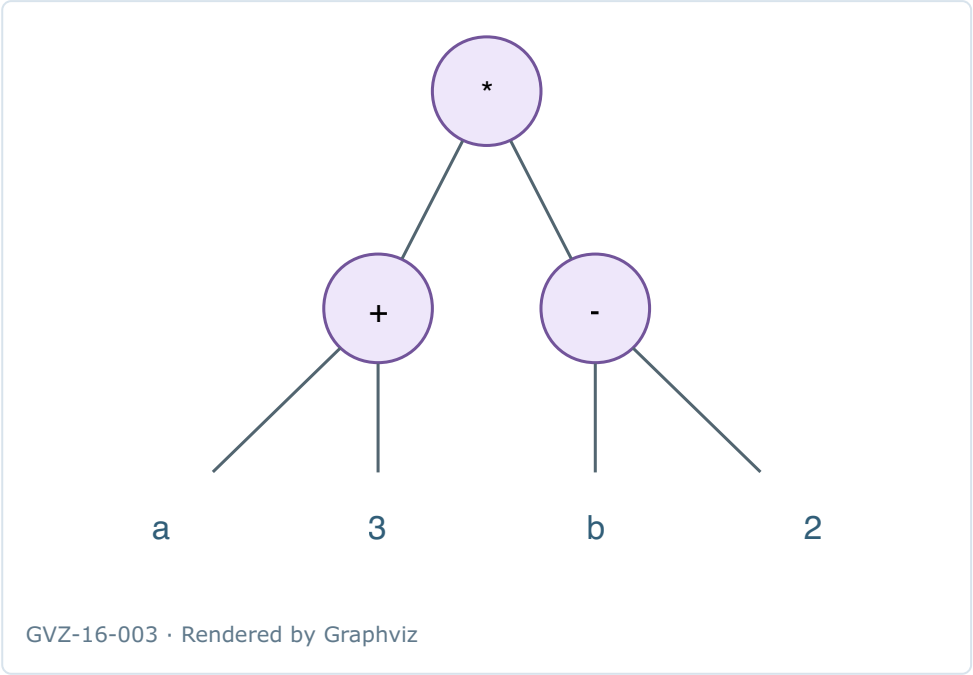
The expression (a + 3) * (b - 2) becomes an ordered tree. Operators are filled circles, while operands are plain labels. ordering=out preserves the declaration order of children, which matters for subtraction. Node identifiers remain unique even when two leaves later display the same text.

Try changing: Swap the two children of subtraction and explain why the expression changes.

DOT source

```
digraph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [ordering=out];
  node [shape=circle, fillcolor="#eee7fa", color="#72549a"];
  multiply [label="*"]; plus [label="+"]; minus [label="-"];
  node [shape=plaintext, style="", fontcolor="#315e78"];
  a [label="a"]; three [label="3"]; b [label="b"]; two [label="2"];
  edge [arrowhead=none];
  multiply -> plus; multiply -> minus;
  plus -> a; plus -> three;
  minus -> b; minus -> two;
}
```

Actual output



Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-16-004

Share prefixes in a trie

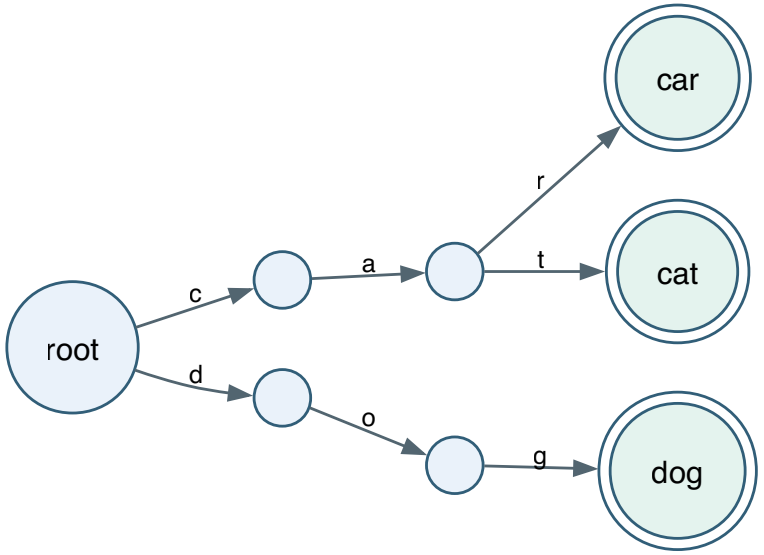
Letters live on edges, so the path from the root spells a word. Double circles mark stored words, not merely intermediate prefixes. The words car and cat share two edges, whereas dog begins on another branch. This is a structural snapshot of a trie; Graphviz does not insert or search the words.

Try changing: Add cart and keep car marked as a complete word as well as a prefix.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=circle, style=filled, label="", width=0.28, height=0.28];
  root [label="root", width=0.65];
  c; ca; d; do;
  car [shape=doublecircle, label="car", fillcolor="#e4f3ee"];
  cat [shape=doublecircle, label="cat", fillcolor="#e4f3ee"];
  dog [shape=doublecircle, label="dog", fillcolor="#e4f3ee"];
  root -> c [label="c"]; c -> ca [label="a"];
  ca -> car [label="r"]; ca -> cat [label="t"];
  root -> d [label="d"]; d -> do [label="o"];
  do -> dog [label="g"];
}
```

Actual output



GVZ-16-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-16-005

Arrange a topic map around a central question

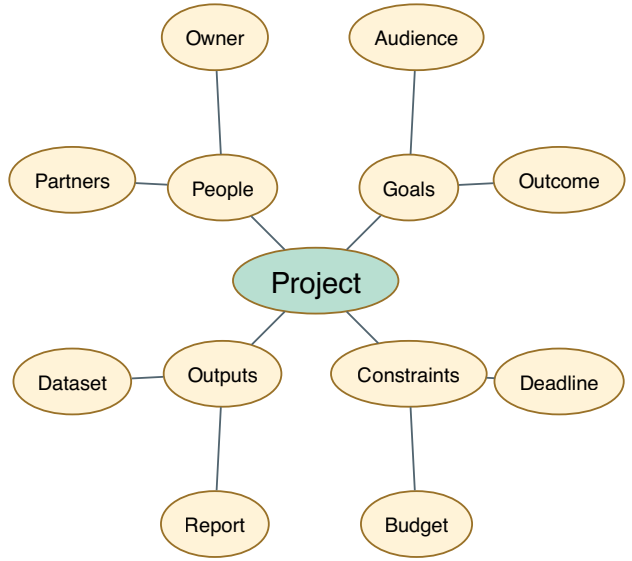
twopi places nodes in radial layers around the named root. Four topic families and their leaves form a mind map rather than a chronological process. The named root controls the center, while ranksep adjusts radial spacing. overlap=true keeps this compact placement, so check label clearance after edits. Angular position does not encode priority.

Try changing: Move the root to Outputs and compare the placement, checking for crowded labels.

DOT source

```
graph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [layout=twopi, root=project, ranksep=1.0, overlap=true];
  node [shape=ellipse, fillcolor="#fff3d8", color="#997128"];
  project [label="Project", fillcolor="#b8dfd1", fontsize=16];
  goals [label="Goals"]; people [label="People"];
  outputs [label="Outputs"]; limits [label="Constraints"];
  outcome [label="Outcome"]; audience [label="Audience"];
  owner [label="Owner"]; partners [label="Partners"];
  dataset [label="Dataset"]; report [label="Report"];
  budget [label="Budget"]; deadline [label="Deadline"];
  project -- {goals people outputs limits};
  goals -- {outcome audience}; people -- {owner partners};
  outputs -- {dataset report}; limits -- {budget deadline};
}
```

Actual output



GVZ-16-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-16-006

Map relationships between two sets

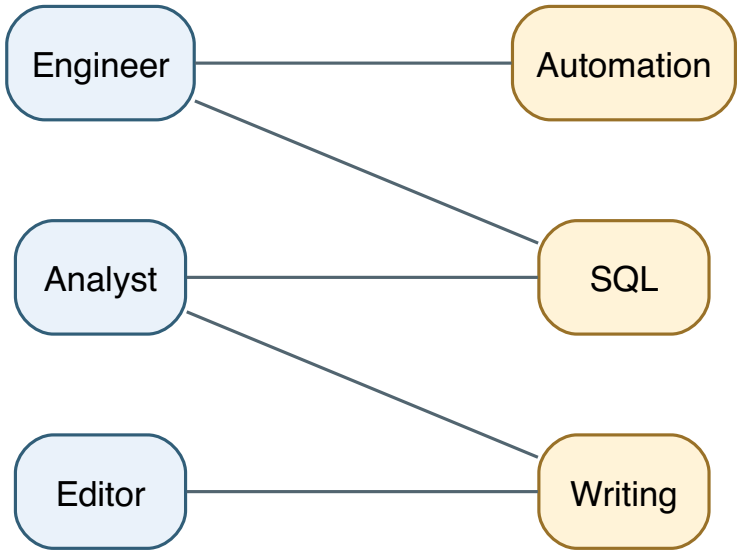
This bipartite graph links roles to capabilities, with no edges inside either set. Equal-rank groups create two columns. Color reinforces the distinction between the sets, but the labels remain sufficient without it. A connection means the capability is relevant to the role; it does not assign a person to a task.

Try changing: Add a role that needs two capabilities and check whether the ordering reduces crossings.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [ranksep=1.4, nodesep=0.45];
  analyst [label="Analyst"]; engineer [label="Engineer"]; editor [label="Editor"];
  node [fillcolor="#fff3d8", color="#997128"];
  sql [label="SQL"]; automation [label="Automation"]; writing [label="Writing"];
  {rank=same; analyst; engineer; editor;}
  {rank=same; sql; automation; writing;}
  edge [arrowhead=none];
  analyst -> sql; analyst -> writing;
  engineer -> sql; engineer -> automation;
  editor -> writing;
}
```

Actual output



GVZ-16-006 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Protocols and coordination

GVZ-17-001

Recognize strings with a finite automaton

This deterministic automaton accepts binary strings ending in 01. q0 has no useful suffix, q1 remembers a trailing 0, and the double-circle q2 is accepting. Every state has exactly one transition for each input symbol. The point marks the initial state; it is not a state that consumes input.

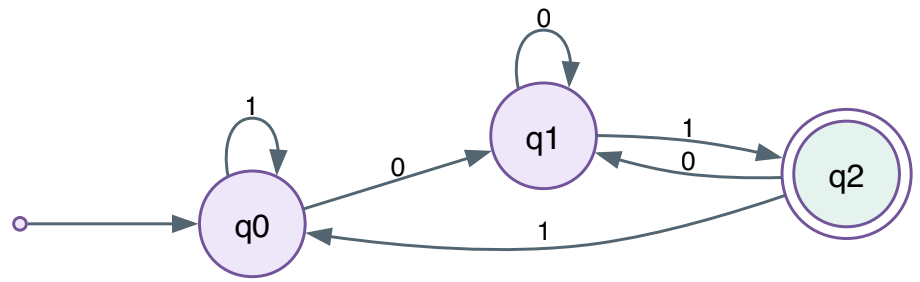
Try changing: Trace 101, 010 and the empty string by hand before changing any transition.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [ranksep=0.8];
  node [shape=circle, style=filled, fillcolor="#eee7fa", color="#72549a"];
  entry [shape=point, label="", width=0.06];
  q0 [label="q0"]; q1 [label="q1"];
  q2 [shape=doublecircle, label="q2", fillcolor="#e4f3ee"];
  entry -> q0;
  q0 -> q0 [label="1"]; q0 -> q1 [label="0"];
  q1 -> q1 [label="0"]; q1 -> q2 [label="1"];
  q2 -> q1 [label="0"]; q2 -> q0 [label="1"];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-17-001 · Rendered by Graphviz

Graphviz · exit 0

GVZ-17-002

Draw a message sequence as a ladder

Vertical dotted lifelines preserve actor identity while horizontal arrows show ordered messages. Same-rank groups align each message row; invisible horizontal edges keep columns in order. Time progresses downward, but the spacing is not a measured duration. This is a small sequence-diagram convention built from ordinary DOT nodes and edges.

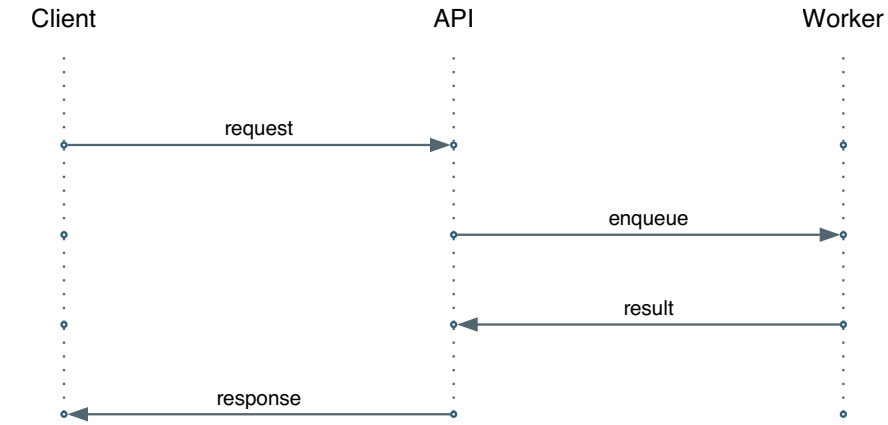
Try changing: Add a timeout response on a new row without reusing an earlier row to imply simultaneous events.

DOT source

```
digraph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [ranksep=0.5, nodesep=0.8];
  node [shape=plaintext, style=""];
  c0 [label="Client", group=c]; a0 [label="API", group=a]; w0 [label="Worker",
group=w];
  node [shape=point, label="", width=0.03, height=0.03];
  {node [group=c]; c1; c2; c3; c4;}
  {node [group=a]; a1; a2; a3; a4;}
  {node [group=w]; w1; w2; w3; w4;}
  {rank=same; c0; a0; w0; c0 -> a0 -> w0 [style=invis];}
  {rank=same; c1; a1; w1; c1 -> a1 -> w1 [style=invis];}
  {rank=same; c2; a2; w2; c2 -> a2 -> w2 [style=invis];}
  {rank=same; c3; a3; w3; c3 -> a3 -> w3 [style=invis];}
  {rank=same; c4; a4; w4; c4 -> a4 -> w4 [style=invis];}
  edge [style=dotted, arrowhead=none, weight=10];
  c0 -> c1 -> c2 -> c3 -> c4;
  a0 -> a1 -> a2 -> a3 -> a4;
  w0 -> w1 -> w2 -> w3 -> w4;
  edge [style=solid, arrowhead=normal, constraint=false];
  c1 -> a1 [label="request"];
  a2 -> w2 [label="enqueue"];
  w3 -> a3 [label="result"];
  a4 -> c4 [label="response"];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-17-002 · Rendered by Graphviz

Graphviz · exit 0

GVZ-17-003

Model a fork and join with a Petri net

Circles are places and narrow bars are transitions. Initially Ready contains one token. Firing the fork creates one token on each branch; the join requires a token from both completed branches. The labels describe that token model, while Graphviz only draws the current structure and marking. It does not simulate transition firing.

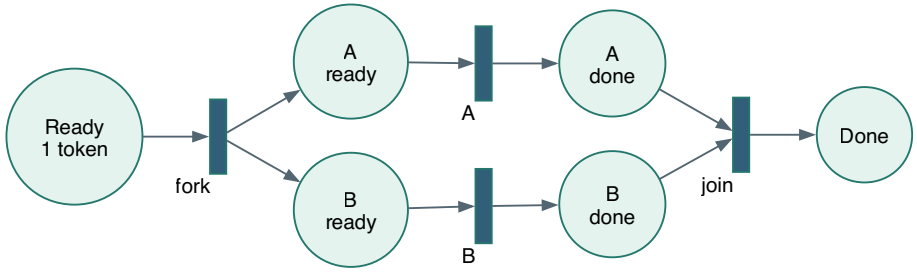
Try changing: Remove one input to the join and explain how the synchronization rule would change.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=circle, style=filled, fillcolor="#e4f3ee", color="#24776d"];
  ready [label="Ready\n1 token"];
  a [label="A\nready"]; b [label="B\nready"];
  a_done [label="A\ndone"]; b_done [label="B\ndone"];
  done [label="Done"];
  node [shape=box, style=filled, fillcolor="#315e78", label="",
        width=0.12, height=0.55, fixedsize=true];
  fork [xlabel="fork"]; ta [xlabel="A"]; tb [xlabel="B"]; join [xlabel="join"];
  ready -> fork;
  fork -> {a b};
  a -> ta -> a_done;
  b -> tb -> b_done;
  {a_done b_done} -> join;
  join -> done;
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-17-003 · Rendered by Graphviz

Graphviz · exit 0

GVZ-17-004

Describe a circuit breaker as three states

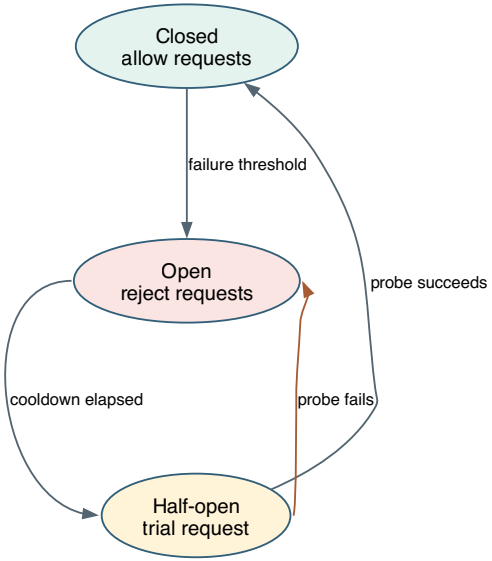
Closed permits normal requests; Open rejects them; Half-open permits a trial. Labeled transitions state the events that move the breaker between those states. The cooldown and failure threshold are policy conditions, not timing behavior provided by Graphviz. Keeping state names inside nodes prevents this from looking like a sequence of service calls.

Try changing: Specify the threshold and trial policy in the labels, then trace a failed probe.

DOT source

```
digraph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [ranksep=1.0];
  node [shape=ellipse, style=filled];
  closed [label="Closed\nallow requests", fillcolor="#e4f3ee"];
  open [label="Open\nreject requests", fillcolor="#fae5e2"];
  half [label="Half-open\ntrial request", fillcolor="#fff3d8"];
  closed -> open [label="failure threshold"];
  open:w -> half:w [label="cooldown elapsed"];
  half -> closed [label="probe succeeds", constraint=false];
  half:e -> open:e [label="probe fails", color="#a75b35"];
}
```

Actual output



GVZ-17-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-17-005

Group related states inside an active region

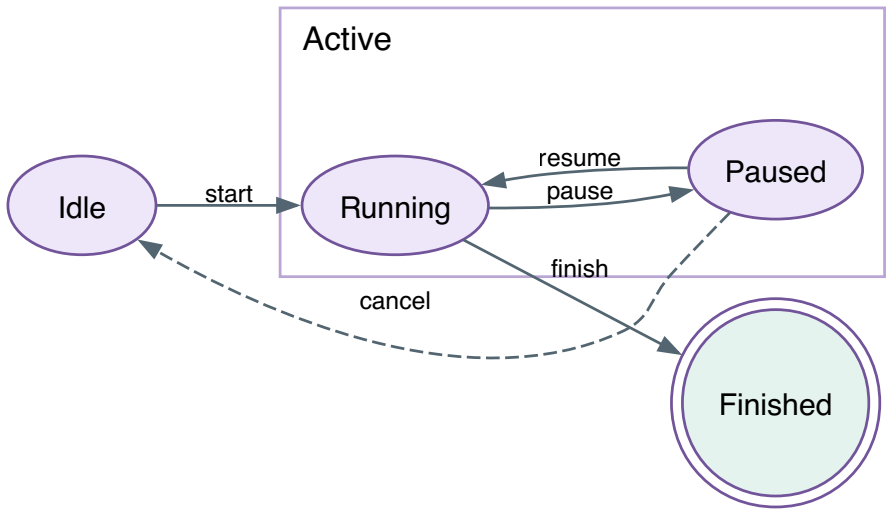
Running and Paused are both active states, so a cluster names their shared region. Only the drawn transitions are permitted in this simplified model: a paused job can resume or cancel, and a running job can finish. The region is an explanatory grouping, not an executable statechart with inherited transitions.

Try changing: Add cancellation from Running explicitly if the application supports it.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=ellipse, style=filled, fillcolor="#eee7fa", color="#72549a"];
  idle [label="Idle"];
  finished [shape=doublecircle, label="Finished", fillcolor="#e4f3ee"];
  subgraph cluster_active {
    label="Active"; color="#b9a6d4"; labeljust=l;
    running [label="Running"]; paused [label="Paused"];
    running -> paused [label="pause"];
    paused -> running [label="resume"];
  }
  idle -> running [label="start"];
  running -> finished [label="finish"];
  paused -> idle [label="cancel", constraint=false, style=dashed];
}
```

Actual output



GVZ-17-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-17-006

Spot a circular resource wait

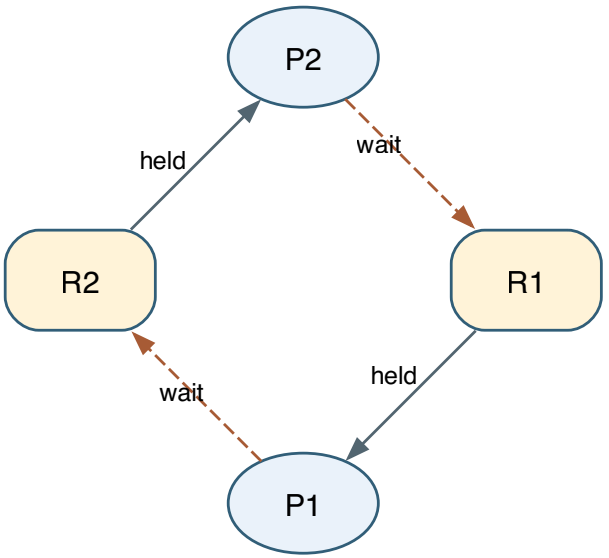
A solid arrow from a resource to a process means allocated; a dashed arrow in the opposite direction means requested. P1 holds R1 and waits for R2, while P2 holds R2 and waits for R1. With exclusive resources and no forced release, this snapshot describes a deadlock. circo makes the circular dependency easy to see; it does not detect deadlocks.

Try changing: Remove one held resource or impose a common acquisition order, then redraw the waiting relationships.

DOT source

```
digraph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [layout=circo, mindist=1.0];
  p1 [shape=ellipse, label="P1"];
  p2 [shape=ellipse, label="P2"];
  r1 [label="R1", fillcolor="#fff3d8"];
  r2 [label="R2", fillcolor="#fff3d8"];
  r1 -> p1 [label="held"];
  p1 -> r2 [label="wait", style=dashed, color="#a75b35"];
  r2 -> p2 [label="held"];
  p2 -> r1 [label="wait", style=dashed, color="#a75b35"];
}
```

Actual output



Graphviz · exit 0

Download: DOT source · SVG diagram

Planning and failure analysis

GVZ-18-001

Highlight a critical path through parallel work

Start and Finish are milestones. The upper branch takes 3 + 5 = 8 days; the lower takes 4 + 2 = 6 days. Assuming both branches can start together and Finish waits for both, the project duration is eight days. Thick teal edges mark the longer branch. These calculations are supplied by the author; DOT labels and edge lengths do not calculate a schedule.

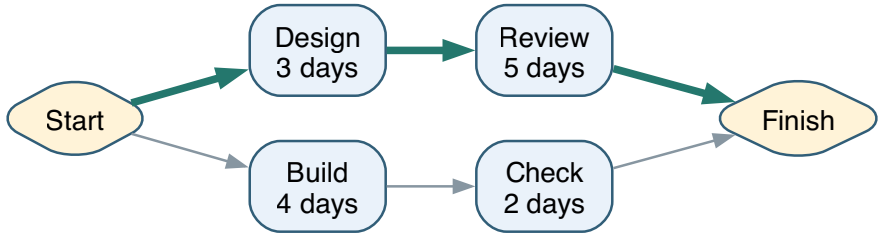
Try changing: Increase Build to seven days and recompute the critical path before changing the highlight.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  start [shape=diamond, label="Start", fillcolor="#fff3d8"];
  finish [shape=diamond, label="Finish", fillcolor="#fff3d8"];
  design [label="Design\n3 days"]; review [label="Review\n5 days"];
  build [label="Build\n4 days"]; check [label="Check\n2 days"];
  start -> design -> review -> finish [color="#24776d", penwidth=3];
  start -> build -> check -> finish [color="#8796a1"];
  {rank=same; design; build;}
  {rank=same; review; check;}
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-18-001 · Rendered by Graphviz

Graphviz · exit 0

GVZ-18-002

Attach deliverables to a milestone roadmap

Diamonds identify three dated milestones. Notes identify the deliverables needed at each milestone, and dashed connectors mean contributes to rather than happens next. The dates are synthetic labels; equal horizontal gaps are not equal calendar intervals. Use a real timeline tool when distance must encode elapsed time.

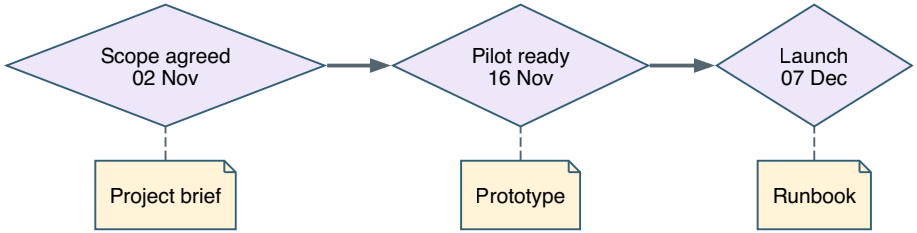
Try changing: Add another deliverable to Pilot without inventing an extra milestone.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=diamond, style=filled, fillcolor="#eee7fa"];
  scope [label="Scope agreed\n02 Nov"];
  pilot [label="Pilot ready\n16 Nov"];
  launch [label="Launch\n07 Dec"];
  scope -> pilot -> launch [penwidth=2];
  node [shape=note, fillcolor="#fff3d8"];
  brief [label="Project brief"];
  prototype [label="Prototype"];
  runbook [label="Runbook"];
  {rank=same; scope; brief;}
  {rank=same; pilot; prototype;}
  {rank=same; launch; runbook;}
  edge [style=dashed, constraint=false, arrowhead=none];
  brief -> scope; prototype -> pilot; runbook -> launch;
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-18-002 · Rendered by Graphviz

Graphviz · exit 0

GVZ-18-003

Separate OR and AND conditions in a fault tree

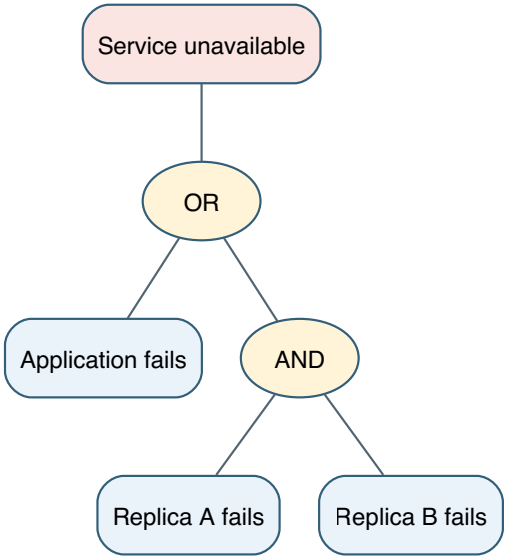
The top event occurs if the application fails OR both replicas fail. Explicitly named gates distinguish those conditions: either child is enough at OR, but both are required at AND. Connectors are arrowless because this tree expands a logical condition rather than scheduling work. No failure probabilities are assumed or computed.

Try changing: Add a third replica under AND and explain the new condition in words.

DOT source

```
digraph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  edge [arrowhead=none];
  outage [label="Service unavailable", fillcolor="#fae5e2"];
  any [shape=ellipse, label="OR", fillcolor="#fff3d8"];
  app [label="Application fails"];
  both [shape=ellipse, label="AND", fillcolor="#fff3d8"];
  a [label="Replica A fails"]; b [label="Replica B fails"];
  outage -> any;
  any -> app; any -> both;
  both -> {a b};
}
```

Actual output



GVZ-18-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-18-004

Draw a signed feedback loop

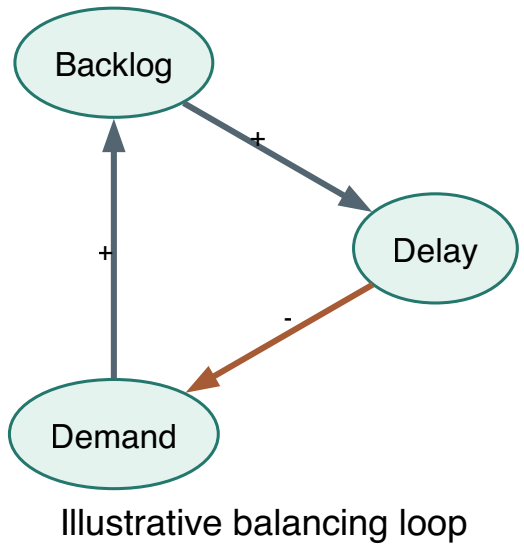
Each node is a quantity, not an activity. A plus sign means an increase tends to increase the next quantity; a minus sign means the opposite, under the assumptions of this illustrative model. More demand increases backlog, backlog increases delay, and delay reduces demand. One negative relationship gives this loop a balancing polarity; no numerical dynamics are simulated.

Try changing: Change the final relationship to a plus and explain how the loop polarity changes.

DOT source

```
digraph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [layout=circo, mindist=1.1, label="Illustrative balancing loop", labelloc=b];
  node [shape=ellipse, fillcolor="#e4f3ee", color="#24776d"];
  demand [label="Demand"]; backlog [label="Backlog"]; delay [label="Delay"];
  demand -> backlog [label="+", penwidth=2];
  backlog -> delay [label="+", penwidth=2];
  delay -> demand [label="-", color="#a75b35", penwidth=2];
}
```

Actual output



GVZ-18-004 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-18-005

Compare alternatives with uncertain durations

The diamond is a choice; the circle is an uncertain outcome. In this synthetic example, Trial takes 30 minutes with probability 0.8 and 90 minutes with probability 0.2, giving an expected time of 42 minutes. Standard takes 50 minutes. The lower mean does not guarantee the shorter run; the diagram keeps that distinction visible.

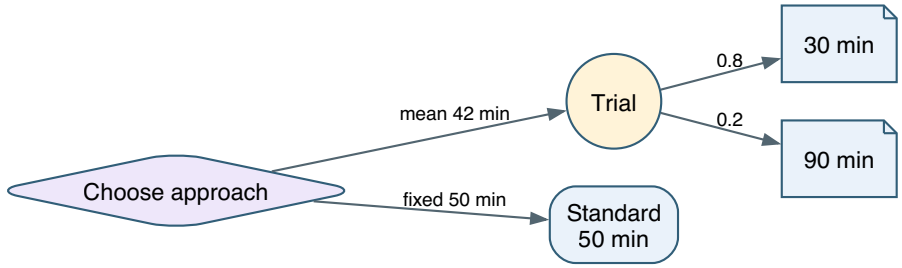
Try changing: Change the slow-case probability to 0.4, update the other probability, and recompute the mean.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  choose [shape=diamond, label="Choose approach", fillcolor="#eee7fa"];
  trial [shape=circle, label="Trial", fillcolor="#fff3d8"];
  standard [label="Standard\n50 min"];
  fast [shape=note, label="30 min"];
  slow [shape=note, label="90 min"];
  choose -> trial [label="mean 42 min"];
  choose -> standard [label="fixed 50 min"];
  trial -> fast [label="0.8"];
  trial -> slow [label="0.2"];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-18-005 · Rendered by Graphviz

Graphviz · exit 0

GVZ-18-006

Connect a goal to actors, behavior and deliverables

An impact map asks who could help achieve a goal, what behavior should change, and which deliverables might enable that change. Shapes distinguish those four levels. Arrows express a proposed contribution chain, not proven causation or task order. This helps separate a desired result from the outputs a team plans to produce.

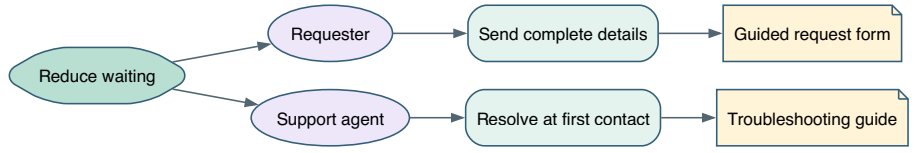
Try changing: Add a second deliverable for one behavior and state how you would check its effect.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  goal [shape=hexagon, label="Reduce waiting", fillcolor="#b8dfd1"];
  node [shape=ellipse, fillcolor="#eee7fa"];
  requester [label="Requester"]; agent [label="Support agent"];
  node [shape=box, fillcolor="#e4f3ee"];
  complete [label="Send complete details"];
  resolve [label="Resolve at first contact"];
  node [shape=note, fillcolor="#fff3d8"];
  form [label="Guided request form"];
  guide [label="Troubleshooting guide"];
  goal -> {requester agent};
  requester -> complete -> form;
  agent -> resolve -> guide;
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-18-006 · Rendered by Graphviz

Graphviz · exit 0

Algorithms and network reasoning

GVZ-19-001

Expose the layers of a dependency DAG

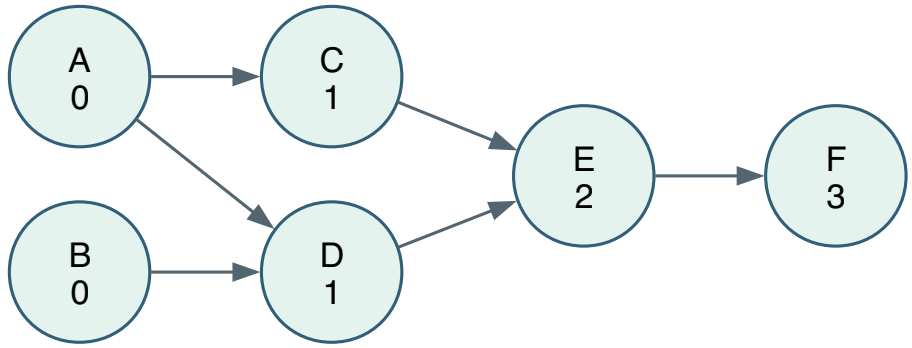
This directed acyclic graph has two initial tasks, two tasks in the next layer, and a final dependency chain. Same-rank groups show a manually assigned valid layering. A and B can be interchanged in a topological ordering, as can C and D once their prerequisites are met. Horizontal placement within a layer is not a dependency.

Try changing: Write two valid topological orders, then add a dependency from C to D and update the layers.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=circle, style=filled, fillcolor="#e4f3ee"];
  a [label="A\n0"]; b [label="B\n0"];
  c [label="C\n1"]; d [label="D\n1"];
  e [label="E\n2"]; f [label="F\n3"];
  {rank=same; a; b;}
  {rank=same; c; d;}
  a -> c; a -> d; b -> d;
  {c d} -> e;
  e -> f;
}
```

Actual output



GVZ-19-001 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-19-002

Highlight a shortest route in a weighted network

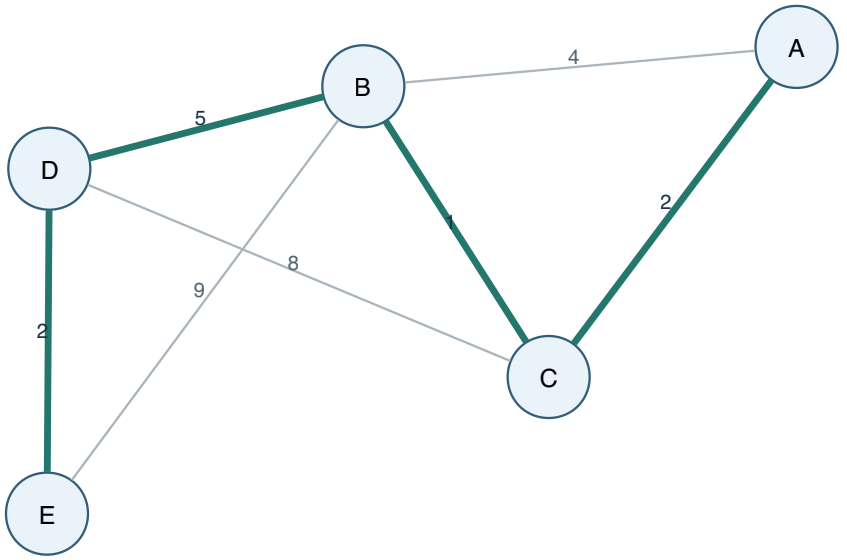
The numbers are nonnegative edge costs. The highlighted route A-C-B-D-E costs 2 + 1 + 5 + 2 = 10, the minimum from A to E in this example. Labels display those costs; they do not ask neato to solve a shortest-path problem. Geometry is chosen for readability and is unrelated to travel distance.

Try changing: Lower the cost of C-D to three and find the new shortest route before updating the highlight.

DOT source

```
graph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [layout=neato, start=17, overlap=false, splines=true];
  node [shape=circle, style=filled, fillcolor="#eaf2fa"];
  A; B; C; D; E;
  edge [color="#aab4bd", fontcolor="#526570"];
  A -- B [label="4"]; C -- D [label="8"]; B -- E [label="9"];
  edge [color="#24776d", penwidth=3, fontcolor="#173147"];
  A -- C [label="2"]; C -- B [label="1"];
  B -- D [label="5"]; D -- E [label="2"];
}
```

Actual output



GVZ-19-002 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-19-003

Keep a minimum spanning tree inside the full network

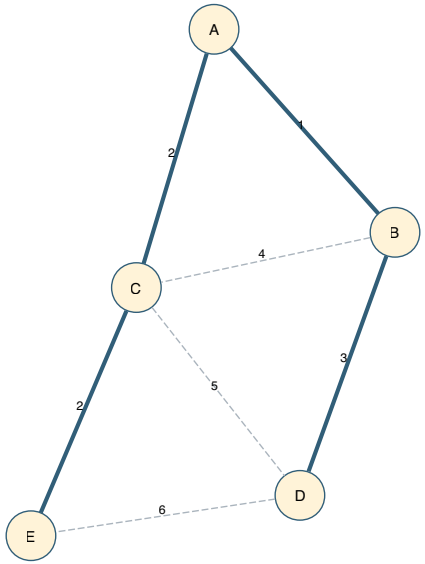
The solid edges connect all five vertices without a cycle, with total cost 1 + 2 + 3 + 2 = 8. The dashed edges preserve the alternatives so the choice remains inspectable. This is a minimum spanning tree for the given costs, chosen before rendering. It minimizes the total connection cost, not every pairwise route.

Try changing: Reduce C-D from five to one and recompute the selected tree.

DOT source

```
graph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [layout=neato, start=6, overlap=false, splines=true];
  node [shape=circle, style=filled, fillcolor="#fff3d8"];
  A; B; C; D; E;
  edge [style=dashed, color="#aab4bd"];
  B -- C [label="4"]; C -- D [label="5"]; D -- E [label="6"];
  edge [style=solid, color="#315e78", penwidth=3];
  A -- B [label="1"]; A -- C [label="2"];
  B -- D [label="3"]; C -- E [label="2"];
}
```

Actual output



GVZ-19-003 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-19-004

Group mutually reachable vertices

A, B and C form one strongly connected component; D and E form another; F is a singleton. Inside each nontrivial component every vertex can reach every other. The arrows between groups only move forward, so collapsing each group would produce a DAG. These groups are authored explicitly, not discovered by the cluster attribute.

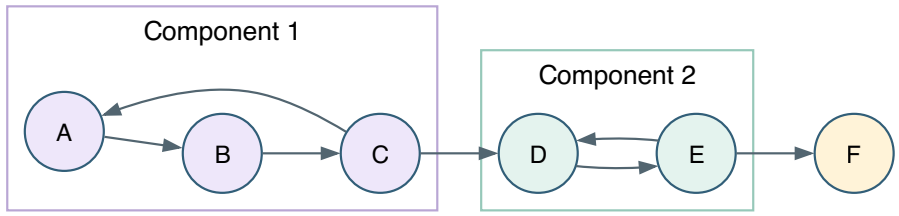
Try changing: Add F to A and determine which components would merge.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  node [shape=circle, style=filled];
  subgraph cluster_one {
    label="Component 1"; color="#b9a6d4";
    node [fillcolor="#eee7fa"];
    A -> B -> C -> A;
  }
  subgraph cluster_two {
    label="Component 2"; color="#96c8b9";
    node [fillcolor="#e4f3ee"];
    D -> E -> D;
  }
  F [fillcolor="#fff3d8"];
  C -> D; E -> F;
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-19-004 · Rendered by Graphviz

Graphviz · exit 0

GVZ-19-005

Color adjacent vertices differently

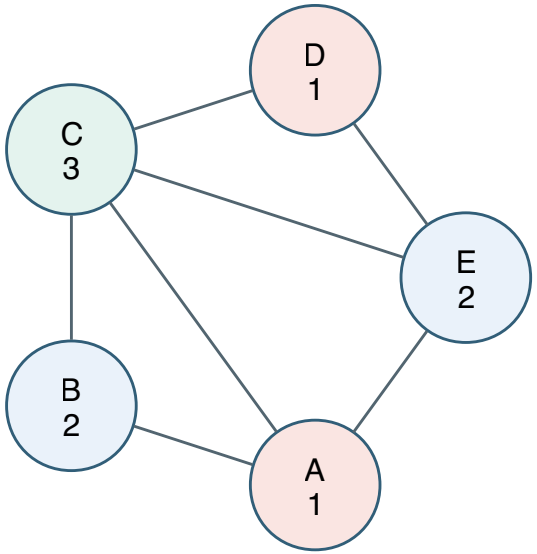
Adjacent vertices must use different classes. Numbers repeat the color classes so the solution is readable in monochrome. A, B and C form a triangle, requiring at least three classes; the displayed assignment uses exactly three for the entire graph. circo arranges the vertices, while the author supplies and checks the coloring.

Try changing: Add an edge between A and D and decide whether three classes still suffice.

DOT source

```
graph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [layout=circo, mindist=0.7];
  node [shape=circle, style=filled];
  A [label="A\n1", fillcolor="#fae5e2"];
  B [label="B\n2", fillcolor="#eaf2fa"];
  C [label="C\n3", fillcolor="#e4f3ee"];
  D [label="D\n1", fillcolor="#fae5e2"];
  E [label="E\n2", fillcolor="#eaf2fa"];
  A -- B -- C -- D -- E -- A;
  A -- C; C -- E;
}
```

Actual output



GVZ-19-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-19-006

Annotate a feasible maximum flow

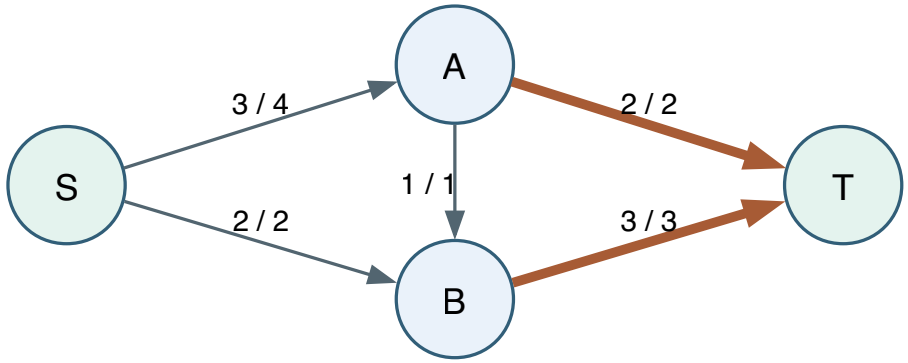
Each label is flow / capacity. Three units enter A and leave as two to T plus one to B; B then sends three units to T. Total flow is five. The two highlighted edges into T have combined capacity five, proving that this feasible flow is maximal. All values are authored calculations; Graphviz does not run a flow algorithm.

Try changing: Raise the capacity of B-T to four and find whether the remaining network can carry a sixth unit.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [ranksep=0.9];
  node [shape=circle, style=filled, fillcolor="#eaf2fa"];
  S [fillcolor="#e4f3ee"]; A; B; T [fillcolor="#e4f3ee"];
  {rank=same; A; B;}
  S -> A [label="3 / 4"];
  S -> B [label="2 / 2"];
  A -> B [label="1 / 1"];
  A -> T [label="2 / 2", color="#a75b35", penwidth=3];
  B -> T [label="3 / 3", color="#a75b35", penwidth=3];
}
```

Actual output



GVZ-19-006 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Data structures and alternative layouts

GVZ-20-001

Connect fields in an entity relationship diagram

HTML-like table labels name fields and expose ports for the relationship. The tee and crow marks, reinforced by the text, mean one Customer may be related to many Orders. The arrow connects the identifier to the foreign-key field rather than to the table center. This compact convention omits optionality and other schema constraints.

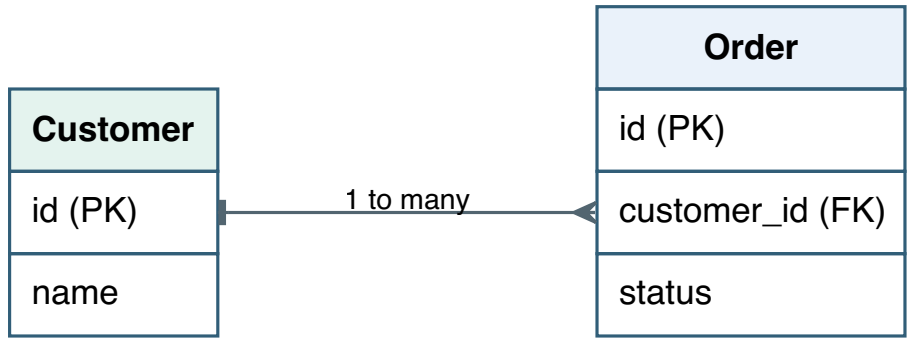
Try changing: Add an OrderLine entity and connect it to the order identifier.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [ranksep=1.1];
  node [shape=plain, style=""];
  customer [label=<<TABLE BORDER="0" CELLBORDER="1" CELLSPACING="0" CELLPADDING="6">
    <TR><TD BGCOLOR="#e4f3ee"><B>Customer</B></TD></TR>
    <TR><TD PORT="id" ALIGN="LEFT">id (PK)</TD></TR>
    <TR><TD ALIGN="LEFT">name</TD></TR>
  </TABLE>>];
  orders [label=<<TABLE BORDER="0" CELLBORDER="1" CELLSPACING="0" CELLPADDING="6">
    <TR><TD BGCOLOR="#eaf2fa"><B>Order</B></TD></TR>
    <TR><TD PORT="id" ALIGN="LEFT">id (PK)</TD></TR>
    <TR><TD PORT="customer" ALIGN="LEFT">customer_id (FK)</TD></TR>
    <TR><TD ALIGN="LEFT">status</TD></TR>
  </TABLE>>];
  customer:id:e -> orders:customer:w [dir=both, arrowtail=tee,
    arrowhead=crow, label="1 to many"];
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-20-001 · Rendered by Graphviz

Graphviz · exit 0

GVZ-20-002

Follow next pointers through a linked list

Record fields separate a stored value from its next pointer. Named ports make each arrow leave the pointer field and enter the next value field. The head reference and NULL terminator describe a singly linked list with three nodes. Record labels use their own braces and port syntax, which differs from HTML-like table labels.

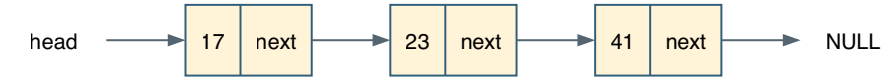
Try changing: Insert a new record between 17 and 23 by changing only the relevant next pointers.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [ranksep=0.55];
  node [shape=record, style=filled, fillcolor="#fff3d8"];
  a [label="{<value> 17|<next> next}"];
  b [label="{<value> 23|<next> next}"];
  c [label="{<value> 41|<next> next}"];
  head [shape=plaintext, style="", label="head"];
  nil [shape=plaintext, style="", label="NULL"];
  head -> a:value:w;
  a:next:e -> b:value:w;
  b:next:e -> c:value:w;
  c:next:e -> nil;
}
```

Download: DOT source · SVG diagram

Actual output



GVZ-20-002 · Rendered by Graphviz

Graphviz · exit 0

GVZ-20-003

Show collisions in a bucket table

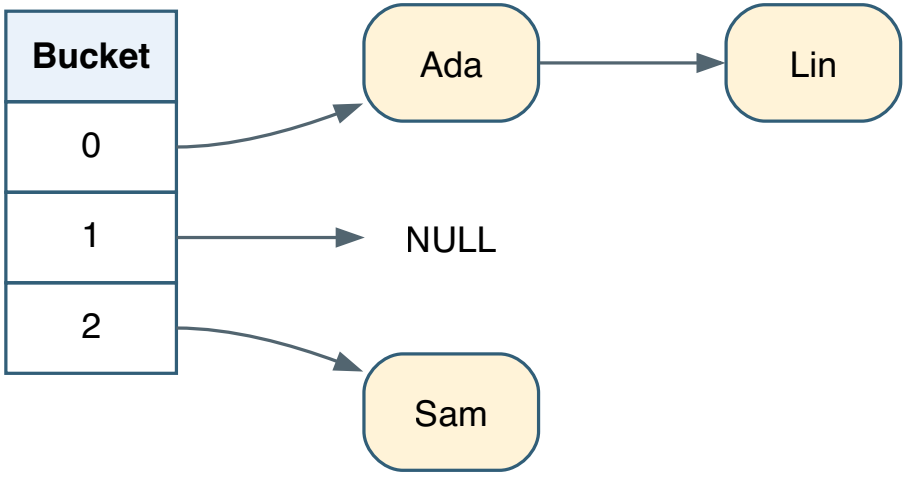
The table provides one port per bucket. Bucket 0 points to a two-entry chain, bucket 1 is empty, and bucket 2 points to one entry. This snapshot illustrates collision handling by chaining; it does not define a hash function or claim these names naturally hash to those bucket numbers.

Try changing: Add a second collision in bucket 2 and keep the empty bucket explicit.

DOT source

```
digraph example {
  graph [rankdir=LR, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [ranksep=0.8];
  buckets [shape=plain, style="", label=<<TABLE BORDER="0" CELLBORDER="1"
CELLSPACING="0" CELLPADDING="7">
    <TR><TD BGCOLOR="#eaf2fa"><B>Bucket</B></TD></TR>
    <TR><TD PORT="b0">0</TD></TR>
    <TR><TD PORT="b1">1</TD></TR>
    <TR><TD PORT="b2">2</TD></TR>
  </TABLE>>];
  node [fillcolor="#fff3d8"];
  ada [label="Ada"]; lin [label="Lin"]; sam [label="Sam"];
  nil [shape=plaintext, style="", label="NULL"];
  buckets:b0:e -> ada -> lin;
  buckets:b1:e -> nil;
  buckets:b2:e -> sam;
  {rank=same; ada; nil; sam;}
}
```

Actual output



Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-20-004

Pin a spatial arrangement with neato

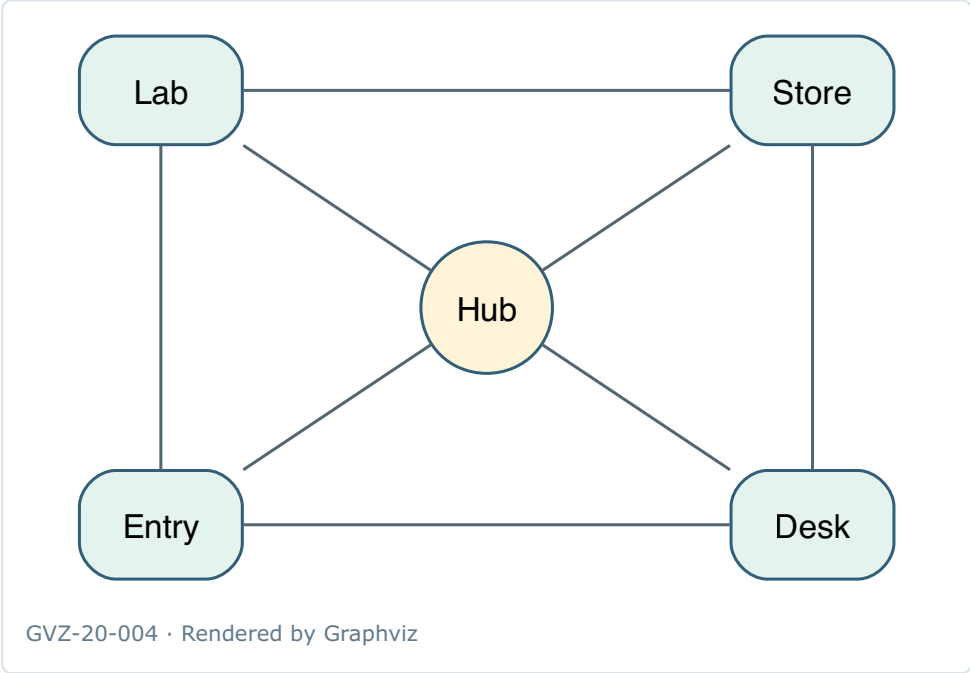
pos supplies node centers in inches for this neato input, and the exclamation mark pins each node. The drawing preserves a deliberately chosen room-like arrangement instead of deriving ranks. overlap=true avoids an overlap-removal pass that could move coordinates, so the author must leave enough space between nodes. The positions are illustrative, not a measured floor plan.

Try changing: Move Hub to 1.5,1.5 and inspect both node clearance and the new edge routes.

DOT source

```
graph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [layout=neato, overlap=true, notranslate=true, splines=true];
  node [shape=box, style="rounded,filled", fillcolor="#e4f3ee"];
  entry [label="Entry", pos="0,0!"];
  desk [label="Desk", pos="3,0!"];
  store [label="Store", pos="3,2!"];
  lab [label="Lab", pos="0,2!"];
  hub [shape=circle, label="Hub", pos="1.5,1!", fillcolor="#fff3d8"];
  entry -- desk -- store -- lab -- entry;
  hub -- {entry desk store lab};
}
```

Actual output



Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-20-005

Arrange clustered collaboration with fdp

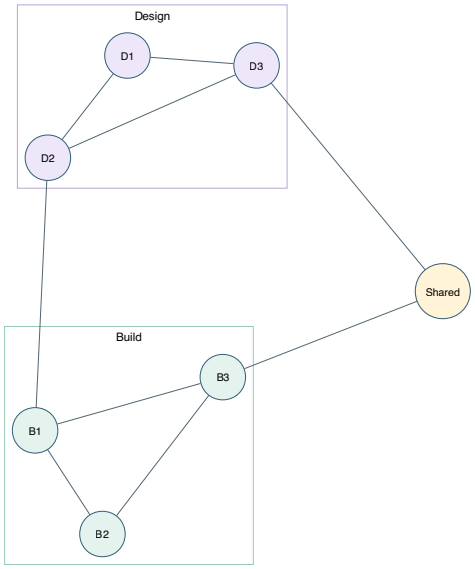
fdp uses a force-directed layout and supports clustered undirected graphs. Two clusters name collaborating groups, while a shared node sits outside them. The integer start value makes the initial layout repeatable for a given setup; it does not guarantee identical geometry across Graphviz versions. Edges mean collaboration and have no chronological direction.

Try changing: Add one cross-group connection and compare how the clusters reposition.

DOT source

```
graph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [layout=fdp, start=7, overlap=false, splines=true];
  node [shape=circle, style=filled, width=0.65];
  subgraph cluster_design {
    label="Design"; color="#b9a6d4";
    node [fillcolor="#eee7fa"];
    D1 -- D2 -- D3 -- D1;
  }
  subgraph cluster_build {
    label="Build"; color="#96c8b9";
    node [fillcolor="#e4f3ee"];
    B1 -- B2 -- B3 -- B1;
  }
  shared [label="Shared", fillcolor="#fff3d8"];
  D2 -- B1;
  D3 -- shared -- B3;
}
```

Actual output



GVZ-20-005 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

GVZ-20-006

Reveal neighborhoods with sfdp

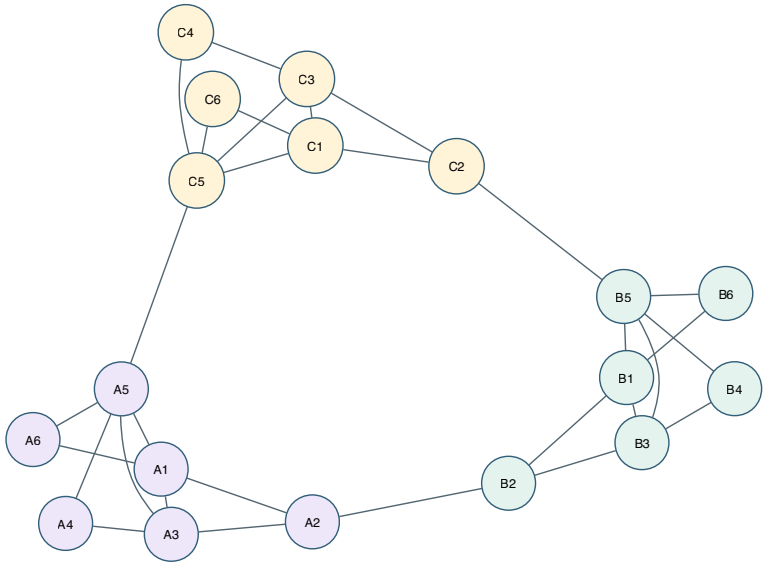
sfdp is a multilevel force-directed engine intended for larger graphs; this small eighteen-node example makes its structure easy to inspect. Dense local connections and sparse bridges separate three neighborhoods. Colors and letter prefixes are supplied group labels, not community detection results. Unlike a workflow, this undirected graph has no start or finish.

Try changing: Add three bridges between the A and C groups and compare their separation.

DOT source

```
graph example {
  graph [rankdir=TB, fontname="Helvetica", fontsize=12];
  node [shape=box, style="rounded,filled", fillcolor="#eaf2fa",
        color="#315e78", fontname="Helvetica", fontsize=11];
  edge [color="#526570", fontname="Helvetica", fontsize=9, arrowsize=0.7];
  graph [layout=sfdp, start=11, overlap=prism, K=0.7, splines=true];
  node [shape=circle, style=filled, width=0.35, height=0.35, fontsize=9];
  {node [fillcolor="#eee7fa"]; A1; A2; A3; A4; A5; A6;}
  {node [fillcolor="#e4f3ee"]; B1; B2; B3; B4; B5; B6;}
  {node [fillcolor="#fff3d8"]; C1; C2; C3; C4; C5; C6;}
  A1 -- A2 -- A3 -- A4 -- A5 -- A6 -- A1;
  A1 -- A3 -- A5 -- A1;
  B1 -- B2 -- B3 -- B4 -- B5 -- B6 -- B1;
  B1 -- B3 -- B5 -- B1;
  C1 -- C2 -- C3 -- C4 -- C5 -- C6 -- C1;
  C1 -- C3 -- C5 -- C1;
  A2 -- B2; B5 -- C2; A5 -- C5;
}
```

Actual output



GVZ-20-006 · Rendered by Graphviz

Graphviz · exit 0

Download: DOT source · SVG diagram

Example and command index

- GVZ-01-001 · Connect three steps
- GVZ-01-002 · Draw an undirected network
- GVZ-01-003 · Branch and join
- GVZ-02-001 · Separate names from labels
- GVZ-02-002 · Set defaults and override a node
- GVZ-02-003 · Explain decisions on edges
- GVZ-03-001 · Align parallel steps
- GVZ-03-002 · Group nodes into clusters
- GVZ-03-003 · Connect to a table cell
- GVZ-04-001 · Arrange a network with neato
- GVZ-04-002 · Arrange a cycle with circo
- GVZ-04-003 · Expand from a root with twopi
- GVZ-05-001 · Mark a clear start and finish
- GVZ-05-002 · Choose exactly one branch
- GVZ-05-003 · Fork work and wait for both branches
- GVZ-05-004 · Show an optional activity
- GVZ-05-005 · Make approval a gate
- GVZ-05-006 · Separate states from activities
- GVZ-06-001 · Bound a retry loop
- GVZ-06-002 · Give a waiting step a timeout
- GVZ-06-003 · Use a fallback with a declared limit
- GVZ-06-004 · Compensate for completed work
- GVZ-06-005 · Make cancellation a separate outcome
- GVZ-06-006 · Quarantine invalid input for repair
- GVZ-07-001 · Expose a handoff between teams
- GVZ-07-002 · Build two vertical swimlanes
- GVZ-07-003 · Turn the lanes horizontally
- GVZ-07-004 · Separate a decision from its owner
- GVZ-07-005 · Distinguish asynchronous messages
- GVZ-07-006 · Escalate with an explicit return path
- GVZ-08-001 · Gate a build on independent checks
- GVZ-08-002 · Promote a release and verify it
- GVZ-08-003 · Show both good and rejected data
- GVZ-08-004 · Handle a poison message
- GVZ-08-005 · Publish documentation with review
- GVZ-08-006 · Keep a human review in the loop
- GVZ-09-001 · Close a service request with confirmation
- GVZ-09-002 · Trace a purchase request to receipt
- GVZ-09-003 · Route defects through rework
- GVZ-09-004 · Track a hiring workflow with explicit exits
- GVZ-09-005 · Coordinate onboarding prerequisites
- GVZ-09-006 · Include learning after restoration
- GVZ-10-001 · Use a small semantic palette
- GVZ-10-002 · Make the design work in monochrome
- GVZ-10-003 · Emphasize the path you want readers to follow
- GVZ-10-004 · Set typography at three levels
- GVZ-10-005 · Control padding and minimum node size
- GVZ-10-006 · Put the legend next to the process
- GVZ-11-001 · Choose a reading direction
- GVZ-11-002 · Separate rank spacing from node spacing
- GVZ-11-003 · Reserve space for a longer dependency
- GVZ-11-004 · Keep branch order stable
- GVZ-11-005 · Use invisible edges as layout scaffolding
- GVZ-11-006 · Anchor the entry and exit ranks

- GVZ-12-001 · Route smooth splines
- GVZ-12-002 · Use segmented polylines
- GVZ-12-003 · Use orthogonal routing carefully
- GVZ-12-004 · Choose where edges meet a node
- GVZ-12-005 · Keep feedback from pulling the main flow
- GVZ-12-006 · Connect cluster boundaries
- GVZ-13-001 · Build an activity card with a subtitle
- GVZ-13-002 · Give inputs and outputs named ports
- GVZ-13-003 · Align multiline text deliberately
- GVZ-13-004 · Label the relationship and its endpoints
- GVZ-13-005 · Attach a note without changing the process
- GVZ-13-006 · Add a title and a scope statement
- GVZ-14-001 · Nest phases inside an overall process
- GVZ-14-002 · Align three stages across two owners
- GVZ-14-003 · Pack independent workflows as small multiples
- GVZ-14-004 · Show detail through a named sub-process
- GVZ-14-005 · Compose a finished approval diagram
- GVZ-14-006 · Compose a process with artifacts and owners
- GVZ-15-001 · Map a service architecture
- GVZ-15-002 · Separate build actions from artifacts
- GVZ-15-003 · Trace the lineage of a derived dataset
- GVZ-15-004 · Show event distribution to independent subscribers
- GVZ-15-005 · Distinguish replicas from independent services
- GVZ-15-006 · Draw dependencies around an interface
- GVZ-16-001 · Draw an organization tree
- GVZ-16-002 · Build a taxonomy with uneven branches
- GVZ-16-003 · Visualize an expression as a syntax tree
- GVZ-16-004 · Share prefixes in a trie
- GVZ-16-005 · Arrange a topic map around a central question
- GVZ-16-006 · Map relationships between two sets
- GVZ-17-001 · Recognize strings with a finite automaton
- GVZ-17-002 · Draw a message sequence as a ladder
- GVZ-17-003 · Model a fork and join with a Petri net
- GVZ-17-004 · Describe a circuit breaker as three states
- GVZ-17-005 · Group related states inside an active region
- GVZ-17-006 · Spot a circular resource wait
- GVZ-18-001 · Highlight a critical path through parallel work
- GVZ-18-002 · Attach deliverables to a milestone roadmap
- GVZ-18-003 · Separate OR and AND conditions in a fault tree
- GVZ-18-004 · Draw a signed feedback loop
- GVZ-18-005 · Compare alternatives with uncertain durations
- GVZ-18-006 · Connect a goal to actors, behavior and deliverables
- GVZ-19-001 · Expose the layers of a dependency DAG
- GVZ-19-002 · Highlight a shortest route in a weighted network
- GVZ-19-003 · Keep a minimum spanning tree inside the full network
- GVZ-19-004 · Group mutually reachable vertices
- GVZ-19-005 · Color adjacent vertices differently
- GVZ-19-006 · Annotate a feasible maximum flow
- GVZ-20-001 · Connect fields in an entity relationship diagram
- GVZ-20-002 · Follow next pointers through a linked list
- GVZ-20-003 · Show collisions in a bucket table
- GVZ-20-004 · Pin a spatial arrangement with neato
- GVZ-20-005 · Arrange clustered collaboration with fdp
- GVZ-20-006 · Reveal neighborhoods with sfdp

Content: CC0 1.0. Structure and publishing framework: CC BY 4.0 by Lukas C. Bossert. Repository source · Publishing framework source · License scope and third-party notices.