

Moket

Moket is a parametric CAD that runs in the browser. You work in it through its MCP server (tools such as `list_projects`, `inspect_document` and `preview_plan`); this skill tells you how to use them well.

Before you start

1. Check that Moket's tools are available. If they are not, tell the user to connect Moket to you first: <https://app.moket.io/help/ai/connectors> explains it for each app.
2. Read `Working with plans` and `Working over MCP` once.
3. Before each kind of work, read its guide below. When Moket is connected, `read_guide` with the guide's name gives the current text; the files here are the version 2026.10.20-a29c23b copy.

How to work

- Find the project and the document with `list_projects`, or make them with `create_project`, `create_part`, `create_assembly` and `create_drawing`. Never ask the user to open a document.
- Look before you change: `inspect_document`, `inspect_assembly` or `inspect_drawing`. Each names the guide for its document.
- Change only through `preview_plan`, with checks for every limit the user gives; it applies at once when it is safe and the user's approvals allow it, otherwise it waits for the user.
- Look for bought parts with `find_library_part` and `find_standard_part` before modeling one.
- Tell Moket's team with `send_feedback` when a tool behaves differently from its description.

Guides

- `Working with plans`: How every change is made: looking first, plans with placeholders and checks, failed previews, approvals and feedback. Read it before your first change in a session.
- `Parts`: Parameters, features and their order, topological names, holes, fillets, shell, draft, sweep, loft, mirror, threads, coils, knurls, configurations, materials and appearance. Read it before you build or change a part.
- `Sketches`: Sketch templates, sketch planes, constraints and dimensions, trim, extend, offset, mirror, fillet and projection, splines and text. Read it before you sketch anything beyond a template.
- `Assemblies`: Components and grounding, joints and their origins, interference, subassemblies, configurations, component parameters, designing a part in context, screws and other standard parts. Read it before you build or change an assembly.
- `Mechanisms`: Closed loops and driven axes, rigid groups, gears, belts, racks and screws, motion studies with collision checks, exploded views. Read it before you make something move.
- `Drawings`: Standards and sheets, views, sections and details, dimensioning a part completely, hole callouts, fits, datums and GD&T, surface texture, title blocks, assembly drawings. Read it before you make or change a drawing.
- `Project parameters`: Sizes several documents share in the project's table, read as `project.<name>`, and what a change of the table reaches. Read it when several documents must agree on a size, or on the project table.
- `Project notes`: The project's Markdown notes: read them before you work, keep what you decided

and checked at the end, and edit them a section at a time. Read it before you work on a project, and at the end of a session that decided or checked something.

- Library and standard parts: Bought components from the component library, modeling one from a datasheet and publishing it, screws, nuts, washers and bearings from the standard parts. Read it before you model anything someone would buy.
- Leather: The Leather Workbench: pieces stacked in layers, stitch lines and their holes through every layer, hole counts, seam allowances, folds and a bifold wallet. Read it before you design leather goods (wallets, card holders, straps).
- Working over MCP: Projects and documents from outside Mocket, plans that wait for the user, and what to do when nothing can open a document. Read it when you reach Mocket over MCP.

Working with plans

Every change in Mocket, by a person, Mocket AI or an agent, is a command on the document's Command Bus. You never edit a document directly: you propose a plan with `preview_plan`, Mocket builds it, measures it and shows it to the user, and it applies when it is safe and the user's approvals allow it. This guide is how to do that well on any document. The guides for parts, sketches, assemblies, mechanisms and drawings say what to put in the plans.

Look before you change

1. Read the document first: `inspect_document` on a part, `inspect_assembly` on an assembly, `inspect_drawing` on a drawing. They give the parameters, the history or the components, what is broken and the revision.
2. Then read what the request involves: `get_feature` for a feature's full parameters, `get_sketch` for a sketch, `query_geometry` for faces and edges with their topological names, `measure` for sizes and distances, `get_view` for a drawing view's edges.
3. When the user says "this" or "these", call `get_selection`: it gives what they selected, with names.
4. Before using a command or feature type for the first time in a conversation, read it with `describe_commands` (`{"types":["core.hole","joint.create"]}`): its JSON Schema and notes are the exact shape Mocket accepts. It answers for the document you call it on: drawing commands on a drawing, assembly commands on an assembly.

A plan

`preview_plan` takes a short summary, steps with labels a person understands, and the commands of each step. All steps are previewed together as one transaction: it must rebuild without errors, or nothing changes.

```
{
  "summary": "Plate 120 x 80 x 6 with four corner holes",
  "steps": [
    {
      "label": "Parameters",
      "commands": [
        {
          "type": "parameter.create",
          "id": "$width",
          "name": "width",
          "expression": "120 mm"
        },
        {
          "type": "parameter.create",
          "id": "$depth",
          "name": "depth",
          "expression": "80 mm"
        },
        {
          "type": "parameter.create",
          "id": "$thickness",
          "name": "thickness",
          "expression": "6 mm"
        }
      ]
    },
    {
      "label": "Plate",
```

```

    "commands": [
      {
        "type": "sketch.template",
        "id": "$outline",
        "name": "Outline",
        "template": {
          "kind": "rectangle",
          "width": "width",
          "depth": "depth"
        }
      },
      {
        "type": "feature.create",
        "id": "$base",
        "featureType": "core.extrude",
        "name": "Plate",
        "params": {
          "sketch": "$outline",
          "operation": "new",
          "body": "$plate",
          "distance": "thickness",
          "direction": "normal"
        }
      }
    ]
  },
  "checks": [
    {
      "label": "Plate volume",
      "quantity": "volume",
      "a": { "kind": "body", "body": "$plate" },
      "min": 57000,
      "max": 58000
    }
  ]
}

```

- Ids and placeholders. New entities need ids. Write placeholders that start with \$ ("base", "holes"); Mocket gives each a UUID and returns the mapping in ids. Use the real ids in later plans. Placeholders work anywhere in the plan: inside topological names ("f(\$base:end)" is the end face of the extrude \$base), in checks ({ "kind": "body", "body": "\$plate" }), as configuration column keys.
- Values have units. Write "24 mm", "45 deg", "0.5 in"; a bare number is in the document's units. Expressions can use parameters: "width - 2 * margin".
- Parameters for what a person would change. Thicknesses, diameters, spacing, counts get named parameters; reuse the document's existing ones instead of repeating a number.
- Names for people. Name the document with document.rename when you make a new one, and give features and components names a person would use ("Mounting holes", not "Hole 3").
- Never invent a topological name. Face and edge names come from query_geometry, measure, get_selection or get_view, or are built from your own placeholders in the forms the feature notes give. Never edit one.

Checks

Add a check for every limit the user states or implies: a minimum wall, an edge distance, a mass, no interference. Mocket measures them on the previewed geometry and shows the results; a plan that fails one waits for the user.

- Parts: distance (between a and b, mm), angle (degrees), radius, length (an edge), area (a face, mm²), volume (a body, mm³), mass (a body, g). Items are { "kind": "body", "body": "<id>" } or

`{"kind":"face"|"edge","name":<topological name>}`.

- Assemblies, without items: interference (overlapping volume, "max":0 for none) and dof ("min":1,"max":1 for exactly one free motion).

Never claim a measurement you did not get from a tool.

When it applies

A previewed plan that rebuilds, passes every check and loses no reference applies at once (its answer says `autoApplied`), unless the user asked to approve that kind of change: every change, deleting or suppressing, or changing what is already there. Anything else waits for the user to click `Apply` (`waitsForApply` says why). So:

- Never preview a plan that does more than the user asked: it may apply right away.
- When the request cannot be met as stated, preview nothing, explain why with the numbers you measured, and offer concrete options.
- After a plan applies, say briefly what changed. When it waits, say what it does and that it waits for the user.
- A new preview replaces the one before it.

When a preview fails

1. Read its problems: each names the step, the command or feature and what went wrong.
2. Fix the plan and preview again. After three failed attempts, stop and explain what blocks you.
3. Never change a value the user asked for, or anything they did not ask to change, to make a plan pass.
4. If a call fails without an answer (a timeout or a network error), the plan may still have applied: inspect the document and compare its revision before you send it again.

Tell Mocket's team

When a tool or command behaves differently from its description or notes, when you need something Mocket cannot do, or when a description misled you, send it once with `send_feedback` (what you did, what happened, what you expected), then go on with the task. Not for the user's own design questions, and never with the user's private data.

Answering the user

Answers are short and in the user's language. Mention values with units. Do not paste JSON or repeat every step: the user sees the plan. Notes in square brackets in the conversation tell you what the user did with a plan (applied, discarded, changed by hand).

Parts

A part is a parametric document: named parameters (expressions with units), an ordered history of features, and the solid bodies the features build. The history rebuilds from the parameters, so a part you build stays editable by hand: build it the way a careful engineer would, with parameters for the sizes that matter and features in a sensible order.

On a part use `inspect_document`, `get_feature`, `get_sketch`, `query_geometry`, `measure` and `get_selection`. Read [Working with plans](#) first.

Parameters

- `parameter.create` with a name and an expression:
`{"type":"parameter.create","id":"$t","name":"wallThickness","expression":"5 mm"}`. Names are identifiers; expressions use units (mm, cm, m, in, ft, deg, rad), other parameters, `pi` and the functions `sqrt`, `abs`, `min`, `max`, `floor`, `ceil`, `round`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan` and `atan2`.
- `parameter.update` changes an expression by id. Every feature that reads the parameter follows.
- Sizes several documents share come from the project's table as `project.<name>` (see [Project parameters](#)); never repeat a number another document owns.

Features

`feature.create` adds a feature after the rollback marker:

`{"type":"feature.create","id":"$f","featureType":"core.extrude","name":"Base","params":{"...}}`. Read a feature type with `describe_commands` before its first use; `feature.update` replaces all params, so send the complete object from `get_feature` with your changes.

Feature: What it does

`core.sketch`: A 2D sketch on a plane; see [Sketches](#). Prefer the `sketch.template` plan helper.

`core.extrude`: Extrudes a sketch's closed regions: operation new (with a new body id), join, cut or intersect (with targets); direction normal, reversed or symmetric; extent through for cuts.

`core.revolve`: Revolves a sketch about a construction line of it, an origin axis or an edge; angle "360 deg" for a full turn.

`core.hole`: Holes at a sketch's points: simple, counterbore or countersink, through or to a depth, tapped with thread.

`core.fillet`, `core.chamfer`: Rounds or bevels edges of one body by name.

`core.shell`: Hollows a body with open faces and a wall thickness.

`core.draft`: Tilts faces by an angle about a neutral plane, for molded parts.

`core.sweep`, `core.loft`: A profile along a path; a solid through profiles.

`core.pattern`: Repeats features linearly or around an axis.

`core.mirror`: Copies features or bodies across a plane.

`core.combine`: Join, cut or intersect bodies.

`core.plane`: A construction plane, offset from a plane or a face.

`core.thread`, `core.coil`, `core.knurl`: Threads on round faces, springs and coils, knurls.

`core.import`: Solids of a STEP file, made by people only.

`feature.delete`, `feature.rename`, `feature.suppress` and `feature.move`

(`{"type":"feature.move","id":<feature>,"after":<feature>,"after":"start" for first}`) change the history;

timeline.setMarker rolls it back for a look. A feature cannot move before what it uses.

Names of faces and edges

Faces and edges are referenced by topological names (`{"kind": "face", "name": "f(...)"}`, `{"kind": "edge", "name": "e(...)"}`) that survive rebuilds: they name the feature and the sketch curve that made each face, not a number. Get them from `query_geometry` (filter by type, size, position or the feature that made them), `measure` or `get_selection`.

In the same plan you can name faces of features you create from their placeholders, in these forms:

- an extrude's start and end faces: `f($base:start)`, `f($base:end)`; a side face from a sketch line: `f($base:side:<line id>)` (the template answer lists line ids);
- an edge between two faces: `e(<face> | <face>)`, the two names sorted.

Never invent any other name or edit one.

Building a part

1. Name it (`document.rename`) and set its material (`document.setMaterial`, for mass) in the first plan.
2. Create the parameters, then the base solid, then the features that cut and add, then the finishing touches (fillets, chamfers) last: a fillet before a cut is often cut away.
3. Put the origin where the part meets others: the face that sits on the base at $Z = 0$, a shaft's axis on Z . Assemblies and drawings are easier for it.
4. Check sizes with `measure` and add checks to the plan for what the user stated.

```
{
  "summary": "Mounting plate 100 x 60 x 8 with four M5 clearance holes",
  "steps": [
    {
      "label": "Name, material and parameters",
      "commands": [
        { "type": "document.rename", "name": "Mounting plate" },
        { "type": "document.setMaterial", "material": "aluminum-6061" },
        {
          "type": "parameter.create",
          "id": "$width",
          "name": "width",
          "expression": "100 mm"
        },
        {
          "type": "parameter.create",
          "id": "$depth",
          "name": "depth",
          "expression": "60 mm"
        },
        {
          "type": "parameter.create",
          "id": "$thickness",
          "name": "thickness",
          "expression": "8 mm"
        },
        {
          "type": "parameter.create",
          "id": "$inset",
          "name": "holeInset",
          "expression": "10 mm"
        }
      ]
    }
  ],
},
```

```

{
  "label": "Plate, its top face at Z = 0",
  "commands": [
    {
      "type": "sketch.template",
      "id": "$outline",
      "name": "Outline",
      "template": {
        "kind": "rectangle",
        "width": "width",
        "depth": "depth",
        "plane": { "kind": "origin", "plane": "XY" }
      }
    },
    {
      "type": "feature.create",
      "id": "$base",
      "featureType": "core.extrude",
      "name": "Plate",
      "params": {
        "sketch": "$outline",
        "operation": "new",
        "body": "$plate",
        "distance": "thickness",
        "direction": "reversed"
      }
    }
  ]
},
{
  "label": "Four holes 10 mm in from the edges",
  "commands": [
    {
      "type": "sketch.template",
      "id": "$corners",
      "name": "Hole centers",
      "template": {
        "kind": "points",
        "points": [
          {
            "x": "-(width / 2 - holeInset)",
            "y": "-(depth / 2 - holeInset)"
          },
          { "x": "width / 2 - holeInset", "y": "-(depth / 2 - holeInset)" },
          { "x": "width / 2 - holeInset", "y": "depth / 2 - holeInset" },
          { "x": "-(width / 2 - holeInset)", "y": "depth / 2 - holeInset" }
        ],
        "plane": { "kind": "origin", "plane": "XY" }
      }
    },
    {
      "type": "feature.create",
      "id": "$holes",
      "featureType": "core.hole",
      "name": "Mounting holes",
      "params": {
        "sketch": "$corners",
        "targets": ["$plate"],
        "type": "simple",
        "diameter": "5.5 mm",
        "extent": "through"
      }
    }
  ]
}
],
"checks": [
  {
    "label": "Plate volume",
    "quantity": "volume",

```

```

    "a": { "kind": "body", "body": "$plate" },
    "min": 47200,
    "max": 47280
  }
]
}

```

A turned part is a profile revolved about its axis. Draw half of its outline on the XZ plane with the axis on Z:

```

{
  "summary": "Shaft Ø20 x 60 stepping down to Ø15 x 40",
  "steps": [
    {
      "label": "Shaft",
      "commands": [
        { "type": "document.rename", "name": "Stepped shaft" },
        { "type": "document.setMaterial", "material": "steel" },
        {
          "type": "sketch.template",
          "id": "$half",
          "name": "Half profile",
          "template": {
            "kind": "profile",
            "corners": [
              { "x": "0 mm", "y": "0 mm" },
              { "x": "10 mm", "y": "0 mm" },
              { "x": "10 mm", "y": "60 mm" },
              { "x": "7.5 mm", "y": "60 mm" },
              { "x": "7.5 mm", "y": "100 mm" },
              { "x": "0 mm", "y": "100 mm" }
            ],
            "plane": { "kind": "origin", "plane": "XZ" }
          }
        },
      ],
    },
    {
      "type": "feature.create",
      "id": "$turn",
      "featureType": "core.revolve",
      "name": "Shaft",
      "params": {
        "sketch": "$half",
        "axis": { "kind": "axis", "axis": "Z" },
        "angle": "360 deg",
        "direction": "normal",
        "operation": "new",
        "body": "$shaft"
      }
    }
  ]
}

```

Holes

A hole drills against its sketch plane's normal from the sketch's points: from XY down (-Z), from XZ toward +Y, from YZ toward -X, from a face into the body. Sketch the points on the face the hole starts from ("plane":{"kind":"face","ref":{"kind":"face","name":<name>}}), any face of any orientation, and the hole goes into the body; no offset plane is needed. From a plane outside the body facing away from it, set "flip": true. A hole that reaches no body fails with HOLE_MISSES_BODY.

- Clearance holes for screws: M3 3.4 mm, M4 4.5 mm, M5 5.5 mm, M6 6.6 mm, M8 9 mm (medium fit).
- Tapped holes: "thread":{"standard":"iso-coarse","size":"M4"} drills the tap drill (3.3 mm for M4) and

puts a cosmetic thread on the wall; give a depth a little longer than the screw's engagement. A lead screw's nut: {"standard":"iso-trapezoidal","size":"Tr12x3"} (drilled to 9 mm), with starts and leftHanded as the screw has them. Bought trapezoidal nuts (DIN 103 hex, the T8 flanged nut) are in find_standard_part.

- Counterbores and countersinks take their diameter and depth or angle (countersinkDiameter, countersinkAngle).
- Patterns of holes: the points of one sketch, or core.pattern over the hole feature.

Fillets, chamfers and finishing

- core.fillet and core.chamfer take edges of one body by name: find them with query_geometry (for example the edges of the top face, or the edges of a given length) and pass {"kind":"edge","name":<name>} for each.
- Fillet and chamfer last. A shell fails when the wall is as thick as a fillet radius or more (SHELL_THICKER_THAN_FILLET): shell before filleting, or keep the wall thinner.
- Mirror copies extrudes, revolves and holes across a plane, and the copies follow the originals (mode features); fillets and chamfers alone cannot be mirrored (MIRROR_EMPTY): mirror the feature they sit on, then fillet the copy. Mode bodies mirrors whole bodies.

Shell, draft, sweep and loft

- Shell hollows a body: pick the faces to open (none for a closed hollow body) and a wall thickness, inside by default. The inner walls are named f(<shell>:shell:<outer face>).
- Draft tilts faces (usually the sides) by an angle about a neutral plane, so a molded part comes out along the pull; faces keep their names. Draft before fillets and shell.
- Sweep moves a closed profile of one sketch along a path: curves of another sketch (lines, arcs and splines joined end to end, drawn so the path starts at the profile) and/or model edges. Sharp corners are mitred; a profile with holes needs a path without sharp corners. A broken path fails with PATH_NOT_CONNECTED naming the gap.
- Loft builds a solid through two or more profiles in order: one closed region per sketch (sketches on construction planes at different heights) or flat faces of bodies; smooth or ruled. Profiles cannot have holes.

Threads, coils and knurls

- core.thread goes on a cylindrical face: standard iso-coarse, iso-fine, unc or unf, or for lead screws and their nuts iso-trapezoidal (Tr12x3) or acme (1/2-10); the size follows the face unless given (pitch picks among sizes of one diameter: Tr16x4 or Tr16x2); the full face or a length. "starts": "4" makes a multi-start thread (lead = starts x pitch: a 3D printer's Tr8x8(P2) is Tr8x2 with 4 starts) and "leftHanded": true a left-hand one; a screw and its nut must agree on both to stay out of interference. A Tr or ACME nut is drilled to d - P (a tapped hole's thread does it). Cosmetic by default (no geometry change, shown on the face, named in STEP, and a library screw in a tapped hole stays out of interference); "modeled": true cuts the helical groove and is slow. side (internal for a nut or tapped bore, external for a screw) follows the face, so set it only to overrule it. class names the tolerance class and defaults to the usual one: 6g/6H for ISO metric, 2A/2B for UNC and UNF, 7e/7H for Tr, 2G for ACME.
- core.coil winds a circle, square or triangle around a sketch circle's axis: turns and pitch, height and

turns, or height and pitch; taper; ground ends. Springs are coils.

- `core.knurl` puts straight or diamond grooves on round faces, cosmetic or modeled (KNURL_HEAVY above 2000 faces).

Configurations

Configurations are named variants of one part (sizes, lengths, features built or suppressed): a table whose columns are parameters and features and whose first row is the default; an empty cell takes the default's value. Build the part once with parameters, then add rows instead of copying the part:

1. `configuration.create` with the columns (`{"kind":"parameter","id":<parameter>}, {"kind":"feature","id":<feature>}`) starts the table with a default row of the current values.
2. `configuration.addRow` adds a named row with the cells that differ (`{"values":{"column id":"10 mm"}}`; placeholders of the same plan work as keys).
3. The part always is its active row: `parameter.update` and `feature.suppress` on a configured column change that row; `configuration.activate` switches rows.

Export (STEP, STL) builds the active row. For a file of each variant the user exports, switches the row (Part properties > Configuration) and exports again; you can switch it with `configuration.activate` when the user asks. In an assembly every component has its own row, so switching the part's changes no assembly.

Name rows as the maker or the standard names the sizes (M3 x 8, MGN12H). `inspect_document` lists the table.

Material and appearance

- `document.setMaterial` sets every body's material (mass, density); `body.setMaterial` one body's. Materials: generic, steel, stainless-steel, aluminum-6061, cast-aluminum, cast-iron, brass, bronze, copper, titanium, epoxy-granite, abs, pla, petg, nylon, acetal, oak, nbr (describe_commands on `document.setMaterial` has the current list).
- Appearance is not geometry and never rebuilds: `document.setAppearance` for the whole part, `body.setAppearance` for a body, `body.setFaceAppearance` for faces by name; the finest one set wins, and without any each body shows its material's default. Set a material for mass and an appearance for looks.

Pictures

`get_view_image` returns a picture of the open document when the user has turned on Send viewport images to AI; otherwise it says so, and you tell the user where to turn it on if a picture would help. Use it to compare the model with a photo or a sketch the user describes, never instead of measuring.

Imported solids

`core.import` steps are made by people (you cannot upload files): imported bodies have no history, their faces are named `i(<step>:<surface>:...)` from their geometry and work like any other reference (sketch on them, cut, fillet, shell). Never change an Import step's file or solids; `IMPORT_FILE_MISSING`, `IMPORT_UNREADABLE`, `IMPORT_NOT_SOLID` and `IMPORT_PARTIAL` are for the user to fix by choosing the file again.

Sketches

A sketch (`core.sketch`) is a 2D drawing on a plane: points, lines, circles, arcs, splines and text, held in place by constraints and dimensions. Its closed regions become profiles for extrude, revolve, sweep and loft, and its points place holes. Dimensions are expressions, so a sketch follows the part's parameters.

Read [Working with plans and Parts](#) first.

Sketch templates

Most sketches are a rectangle, circles, hole points, a polygon outline or a slot. The `sketch.template` plan helper writes them fully constrained, with every size an expression:

```
{
  "type": "sketch.template",
  "id": "$sk",
  "name": "Base",
  "template": {
    "kind": "rectangle",
    "width": "width",
    "depth": "depth",
    "plane": { "kind": "origin", "plane": "XY" }
  }
}
```

Template: Shape

`{"kind":"rectangle","width":...,"depth":...}`: Centered on the origin; its lines are bottom, right, top, left.

`{"kind":"circles","diameter":"6 mm","centers":[{"x":"0 mm","y":"pitch"}]}`: Circles of one diameter.

`{"kind":"points","points":[{"x":"-30 mm","y":"0 mm"}]}`: Points, for hole centers.

`{"kind":"profile","corners":[{"x":...,"y":...},...],"axis":[[0,0],[0,10]]}`: A closed polygon with every corner dimensioned from the origin; the optional axis is a construction line for a revolve.

`{"kind":"slot","from":{..},"to":{..},"width":"6 mm"}`: A slot between two arc centers; its lines are top and bottom, its arcs start and end.

`{"kind":"polygon","center":{..},"sides":6,"diameter":"10 mm"}`: A regular polygon inscribed in a construction circle, its first side horizontal at the bottom (a hex's flats are $\text{diameter} * \cos(30 \text{ deg})$ apart).

The answer lists the entity ids each template made (sketches in the preview), for later plans that constrain, trim or reference them.

Sketch planes

plane is one of:

- `{"kind":"origin","plane":"XY" | "XZ" | "YZ"}`, the default XY. XY: sketch x is the part's X, y its Y, the normal +Z. XZ: x is X, y is Z, the normal -Y. YZ: x is Y, y is Z, the normal +X. Extrudes go along the normal (reversed against it), holes against it.
- `{"kind":"face","ref":{"kind":"face","name":<topological name>}}`: a planar face of a body, from `query_geometry` or one of your own placeholders (`f($base:end)`). The sketch follows the face when it moves. Its origin is the part's origin dropped onto the face's plane, sketch x is the part's X laid on the face (the part's Y when the face faces mostly along X), and sketch y is the normal crossed with x, so it depends on which way the face faces:

The face faces Sketch x Sketch y			
----- ----- -----			
+Z	X	Y	
-Z	X	-Y	
+Y	X	-Z	
-Y	X	Z	
+X	Y	Z	
-X	Y	-Z	

So hole centers on the top of a flange that faces +Y take the part's Z with its sign changed as y.

- {"kind": "plane", "feature": "<core.plane id>": a construction plane, offset from a plane or a face. The offset goes along the base's normal: from XZ toward -Y, so a plane at Y = 20 mm is XZ with offset -20 mm.

sketch.setPlane moves a sketch to another plane.

Drawing by hand

For shapes no template makes, create a core.sketch with its entities and constraints. Points carry a first guess of their position in mm; the solver moves them to satisfy the constraints. Fix one point (or constrain it to the origin) and dimension the rest until the sketch has no degrees of freedom left (get_sketch says how many remain).

```
{
  "summary": "Triangular gusset 40 x 30, 5 mm thick",
  "steps": [
    {
      "label": "Gusset",
      "commands": [
        { "type": "document.rename", "name": "Gusset" },
        {
          "type": "feature.create",
          "id": "$tri",
          "featureType": "core.sketch",
          "name": "Triangle",
          "params": {
            "plane": { "kind": "origin", "plane": "XY" },
            "entities": [
              { "id": "$p1", "kind": "point", "x": 0, "y": 0, "fixed": true },
              { "id": "$p2", "kind": "point", "x": 40, "y": 0 },
              { "id": "$p3", "kind": "point", "x": 0, "y": 30 },
              { "id": "$bottom", "kind": "line", "start": "$p1", "end": "$p2" },
              { "id": "$side", "kind": "line", "start": "$p1", "end": "$p3" },
              { "id": "$slope", "kind": "line", "start": "$p2", "end": "$p3" }
            ],
            "constraints": [
              { "id": "$h", "kind": "horizontal", "line": "$bottom" },
              { "id": "$v", "kind": "vertical", "line": "$side" },
              {
                "id": "$w",
                "kind": "distance",
                "a": "$p1",
                "b": "$p2",
                "value": "40 mm"
              },
              {
                "id": "$t",
                "kind": "distance",
```

```

        "a": "$p1",
        "b": "$p3",
        "value": "30 mm"
    }
  ],
},
{
  "type": "feature.create",
  "id": "$plate",
  "featureType": "core.extrude",
  "name": "Gusset",
  "params": {
    "sketch": "$tri",
    "operation": "new",
    "body": "$body",
    "distance": "5 mm",
    "direction": "normal"
  }
}
]
}
]
}

```

Constraints: coincident, horizontal, vertical, horizontalPoints, verticalPoints, parallel, perpendicular, equalLength, equalRadius, tangent, pointOnLine, pointOnCircle, midpoint, symmetric. Dimensions, with a value expression: distance, horizontalDistance, verticalDistance, pointLineDistance, radius, diameter, angle, lineOffset, radiusDifference.

Commands on an existing sketch: sketch.add (entities and constraints), sketch.remove, sketch.constrain, sketch.setDimension (a dimension's value), sketch.move, sketch.setConstruction, sketch.setFixed. Construction geometry ("construction": true) guides the shape and makes no profile.

Sketch tools

Commands on a sketch's stored geometry:

- sketch.trim: a curve and a point on the piece to remove; it is cut back to the curves that cross it.
- sketch.extend: a free end to the next curve.
- sketch.offset: curves, a distance expression and its value in mm, side inside, outside or both; linked offsets follow the original. A whole loop is the usual input.
- sketch.mirror: curves and a line; the copies stay symmetric.
- sketch.fillet: a corner point where two lines meet and a radius; the corner stays as a construction point.
- sketch.project: model edges by topological name with their shape on the sketch plane; the rebuild moves them with the edge, and a lost edge leaves SKETCH_PROJECTION_DANGLING. Its ids go in order, per edge: the projection, then its fixed entities (a line: two points and the line; a circle: center and circle; an arc: center, start, end and arc). With "ids":["\$proj","\$c","\$circle"] a later sketch.offset in the same plan takes ["\$circle"].

Slots and regular polygons drawn by hand are lines and arcs added with sketch.add, with tangent, equal and point-on-circle constraints.

Splines

A spline is a sketch entity over ordinary points: by fit it passes through them, by control it is shaped by them (add construction lines between control points to constrain its tangents). Constraints go on its points, not on the curve. `sketch.setSpline` switches between fit and control. Splines work in profiles and in sweep paths.

Text

Text is a sketch entity at a point: an expression height (of a capital letter) and angle, font noto-sans, noto-sans-mono or inter, bold, align left, center or right. Its outlines are closed regions: extrude them to emboss, or cut to engrave. `{name}` in the text is a parameter's value and `{project.name}` a project parameter's, so a part number or a size follows the part. `sketch.setText` changes it.

```
{
  "summary": "Name plate 60 x 20 x 3 with MOKET engraved 0.5 mm deep",
  "steps": [
    {
      "label": "Plate, its top face at Z = 0",
      "commands": [
        { "type": "document.rename", "name": "Name plate" },
        {
          "type": "sketch.template",
          "id": "$outline",
          "name": "Outline",
          "template": {
            "kind": "rectangle",
            "width": "60 mm",
            "depth": "20 mm"
          }
        }
      ],
      {
        "type": "feature.create",
        "id": "$base",
        "featureType": "core.extrude",
        "name": "Plate",
        "params": {
          "sketch": "$outline",
          "operation": "new",
          "body": "$plate",
          "distance": "3 mm",
          "direction": "reversed"
        }
      }
    ]
  },
  {
    "label": "Engraving",
    "commands": [
      {
        "type": "feature.create",
        "id": "$label",
        "featureType": "core.sketch",
        "name": "Label",
        "params": {
          "plane": { "kind": "origin", "plane": "XY" },
          "entities": [
            { "id": "$at", "kind": "point", "x": 0, "y": -4, "fixed": true },
            {
              "id": "$text",
              "kind": "text",
              "text": "MOKET",
              "font": "inter",
              "bold": true,
              "position": "$at",
              "height": "8 mm",
              "angle": "0 deg",
            }
          ]
        }
      }
    ]
  }
}
```

```

        "align": "center"
      }
    ],
    "constraints": []
  }
},
{
  "type": "feature.create",
  "id": "$engrave",
  "featureType": "core.extrude",
  "name": "Engrave",
  "params": {
    "sketch": "$label",
    "operation": "cut",
    "targets": ["$plate"],
    "distance": "0.5 mm",
    "direction": "reversed"
  }
}
]
}
]
}

```

Assemblies

An assembly is a document of its own that puts parts of the project together. Components are instances of parts (or of other assemblies) at their latest revision; one is grounded, and joints between two components place the rest. Positions come from the solver and the parts, never from you: you say which faces meet and how they may move, and Mocket works out where everything goes.

On an assembly use `inspect_assembly`, `query_geometry` with a component, `measure`, `check_interference` and `check_motion`; the part tools answer only on parts. Read *Working with plans* first, and *Mechanisms* when something must move in a loop, by a gear or along a study.

Components

- `component.insert` inserts components of a part or an assembly of the project, a standard part (`find_standard_part`) or a library version (`find_library_part`):
`{"type":"component.insert","part":<id>,"count":2,"ids":["$a","$b"]}`. "name" is for one component (count 1); name several with `component.update` after inserting them.
- The first component of an empty assembly is grounded unless you say otherwise; ground the base part first. Others are free until a joint places them; give a free one a placement (mm and a unit quaternion `[x,y,z,w]`) only when it has no joint.
- `component.update` changes a component's name, placement, grounded, suppressed, configuration, appearance, parameters (below) and, for a subassembly, flexible and positions.
`component.replacePart` moves components to another part and keeps their joints by face names.
`component.delete` deletes one with its joints.

Joints

`joint.create` connects two origins:

`{"type":"joint.create","id":"$j","joint":{"type":"revolute","a":<origin>,"b":<origin>}}`.

- Types: fixed (no motion), revolute (turns: rotation), slider (slides: translation), cylindrical (both), ball (rotation, rotationX, rotationY about one point), planar (translationX, translationY, rotation), pin-slot (rotation and a slide along X, translationX). Another axis key on a type is an error.
- An origin is a face or an edge of a component by its topological name, from `query_geometry` with that component: `{"component":<id>,"ref":{"kind":"face","body":<body id>,"name":<name>},"use":"center"}`, or the component's origin: `{"component":<id>,"ref":"origin"}`.
- "use":"center" on a planar face gives its centroid and outward normal (the centroid of the face as it is: holes in it can move it off the middle; use a component's origin or a circular edge for an exact point); "use":"axis" on a cylindrical, conical or toroidal face or a circular edge gives the axis, centered on the face; on a line edge, its midpoint and direction.
- The joint holds when side a's frame sits on side b's, Z axes opposed (faces meet face to face) unless flip makes them agree (flush, or a shaft standing on a face instead of sinking into it). offset moves along and angle turns about side b's Z. `offsetX` and `offsetY` move the joint's zero across Z, along side b's X and Y: positions and limits count from there, so a carriage's travel can start at a rail's end ("`offsetX`":-150 mm" on a 300 mm rail whose origin is in its middle) without moving the rail's origin.
- A frame's X is the part's X laid flat on the plane across its Z (the part's Y when Z is along X), so a planar or pin-slot joint slides along it; a component's origin has the part's own axes.

- The component nearer the ground stays; the other moves.
- Axes take a position and limits ({ "rotation": { "position": "30 deg", "limits": { "min": "-90 deg", "max": "90 deg" } } }). Positions, offsets and limits are expressions and may use the assembly's parameters (parameter.create on the assembly). An axis whose limits are equal is locked: held there, with no degree of freedom.
- A joint whose face is gone is dangling and moves nothing: reattach it with joint.update (new a or b), or delete it.

A shaft standing on a plate, turning about its axis:

```
{
  "summary": "Shaft standing on the plate's top face, free to turn",
  "steps": [
    {
      "label": "Components",
      "commands": [
        { "type": "document.rename", "name": "Shaft on plate" },
        {
          "type": "component.insert",
          "part": "<Mounting plate>",
          "ids": ["$plate"]
        },
        {
          "type": "component.insert",
          "part": "<Stepped shaft>",
          "ids": ["$shaft"]
        }
      ]
    },
    {
      "label": "The shaft's origin on the middle of the plate's top face",
      "commands": [
        {
          "type": "joint.create",
          "id": "$turn",
          "joint": {
            "type": "revolute",
            "a": { "component": "$shaft", "ref": "origin" },
            "b": {
              "component": "$plate",
              "ref": {
                "kind": "face",
                "body": "<Mounting plate.$plate>",
                "name": "f(<Mounting plate.$base>:start)"
              },
              "use": "center"
            }
          },
          "flip": true
        }
      ]
    }
  ],
  "checks": [
    { "label": "Nothing overlaps", "quantity": "interference", "max": 0 },
    { "label": "Only the shaft turns", "quantity": "dof", "min": 1, "max": 1 }
  ]
}
```

Values in angle brackets stand for ids you get from earlier answers: <Mounting plate> is that part's document id, <Mounting plate.\$base> the id its plan gave \$base. In your own plans, write the real ids, or the names query_geometry returns.

Checking

- Read placements back with `inspect_assembly`: each component's position, how it is placed (ground, joint or stored placement), every joint's status and free axes, the degrees of freedom, mass and center of mass, and the parts list.
- Check overlaps with `check_interference` after placing components, and add plan checks `{"quantity":"interference","max":0}` and `{"quantity":"dof","min":1,"max":1}` (the motion you expect). Touching faces do not count, nor does the ring between a screw and its tapped hole when both carry a thread of the same size and pitch on one axis (standard screws do; give a stud of your own part a `core.thread`, or it counts). A press fit the user wants is marked with `interference.markIntended`.
- `check_interference` with positions checks a pose without changing the assembly: `{"positions":[{"joint":<id>,"axis":"translation","value":"120 mm"}]}`, for the ends of travel.

Subassemblies

An assembly of the project can be a component (`component.insert` with its id). It is rigid by default and moves as one body; `"flexible": true` makes its joints part of this solve, and positions set its free axes from here. Joint origins and `query_geometry` reach a part inside it with path (the components from the subassembly down to the part, as `inspect_assembly` lists them in the subassembly's inside). An assembly cannot contain itself.

Configurations and component parameters

- A part with configurations (its rows are under part on its first component in `inspect_assembly`) is inserted or switched per component with `"configuration":<row id>`; the parts list counts part and row, for example Socket screw M3 x 10.
- A component can set its part's parameters for itself: `component.update` with `"parameters":{"length":"bedLength - 160 mm"}`, by the part's parameter names, as expressions of the assembly (its parameters and `project.<name>`). Use it for a rail, a lead screw or a shaft that takes the length this assembly needs, instead of copying the part or adding a configuration row per length. `inspect_assembly` lists a part's parameters (`partParameters`) on its first component; the parts list counts other values as other items ("Shaft, length 110 mm"); `partsOnly` counts through subassemblies.

Designing a part in context

A part that must fit the parts around it (a cover on a plate, a seat on a bearing, a bracket on a rail's holes) can be designed in the assembly: sketch on another component's face, project its edges, put a plane on it, or extrude toFace up to it. Use this for fits that follow other parts; use project parameters for sizes the user chooses.

- The part must be a component of the assembly first (insert it, New Part in the studio). Then, in a plan on the part, `context.create` names the assembly, the part's own component in it, and references to faces or edges of other components: a source with the component path in the assembly and the face or edge of that component's part, by the names `query_geometry` gives for that component.
- Leave out `copy` and `capturedAt`: Mocket reads them from the assembly as it is now. The part keeps

the copies and builds from them alone, in any assembly or drawing.

- Use a reference by its id: a sketch plane {"kind":"context","ref":<id>} (also as a sketch template's plane), a projection {"context":<id>,"shape":...} in sketch.project (its ids name the projection first, then its entities: for a circle ["\$proj","\$center","\$circle"], so offset \$circle), a core.plane base, or an extrude's "extent":"toFace","toFace":{"kind":"context","ref":<id>}. Only planar faces become planes; only straight and circular edges project.
- When the components it was designed against move or change, inspect_assembly shows the component's context as stale (CONTEXT_STALE); {"type":"context.update","context":<id>} in a plan on the part takes fresh copies, one new version. A source that is gone is lost: the part keeps building; reattach it with context.update and a new source, or change the features that use it.

Insert the part into the assembly, grounded where it goes:

```
{
  "summary": "A cover pad in the assembly, at the plate's origin",
  "steps": [
    {
      "label": "Insert",
      "commands": [
        {
          "type": "component.insert",
          "part": "<Cover pad>",
          "ids": ["$pad"],
          "grounded": true
        }
      ]
    }
  ]
}
```

Then, on the part, a Ø12 pad standing on the plate's top face 25 mm from its middle:

```
{
  "summary": "Pad on the plate's top face, designed in the assembly",
  "steps": [
    {
      "label": "The plate's top face from the assembly",
      "commands": [
        { "type": "document.rename", "name": "Cover pad" },
        {
          "type": "context.create",
          "id": "$ctx",
          "assembly": "<Shaft on plate>",
          "component": "<Shaft on plate.$pad>",
          "refs": [
            {
              "id": "$top",
              "source": {
                "path": ["<Shaft on plate.$plate>"],
                "ref": {
                  "kind": "face",
                  "body": "<Mounting plate.$plate>",
                  "name": "f(<Mounting plate.$base>:end)"
                }
              }
            }
          ]
        }
      ]
    },
    {
      "label": "A pad on it",
      "commands": [
        {

```

```

    "type": "sketch.template",
    "id": "$circle",
    "name": "Pad",
    "template": {
      "kind": "circles",
      "diameter": "12 mm",
      "centers": [{ "x": "25 mm", "y": "0 mm" }],
      "plane": { "kind": "context", "ref": "$top" }
    }
  },
  {
    "type": "feature.create",
    "id": "$boss",
    "featureType": "core.extrude",
    "params": {
      "sketch": "$circle",
      "operation": "new",
      "body": "$body",
      "distance": "4 mm",
      "direction": "normal"
    }
  }
]
}
]
}

```

Screws, nuts, washers and bearings

These come from the built-in ISO and ANSI library, never modeled. `find_standard_part` (`{ "query": "socket head M4" }`) gives a part's id, its rows (one configuration per size and length) and where its origin is. Insert with `component.insert { "part": <id>, "configuration": <row id>, and "appearance" for a finish.`

- Screws stand on their axis with the origin under the head and +Z toward the head (countersunk and set screws: at the top face); washers and nuts have their origin on the bottom face with +Z through them.
- To fasten a hole: a joint from the screw's origin to the mouth edge of the hole the head sits on (`"use": "axis"`, the circular edge where the clearance hole of the top part meets its outer face, from `query_geometry`), flipped or not so the head stays outside (check with `inspect_assembly`); washers and nuts fixed to the screw's origin with an offset. Make it revolute with the spin locked by equal limits (`"rotation": { "position": "0 deg", "limits": { "min": "0 deg", "max": "0 deg" } }`), as the user's Fasten tool does: a fastened screw adds no degree of freedom, and an assembly bolted together has 0. Screws fastened before 2026-10-17 can still spin (one degree of freedom each) until their limits are set.
- Pick the size the hole is made for (a Ø4.5 clearance or a Ø3.3 tapped hole takes M4) and a length that engages about 1.5 diameters of thread without bottoming out. The user's Fasten tool does all this from a click on a hole.

```

{
  "type": "joint.create",
  "id": "$screw1",
  "joint": {
    "type": "revolute",
    "a": { "component": "$screw", "ref": "origin" },
    "b": {
      "component": "<bracket component>",
      "ref": {
        "kind": "edge",
        "body": "<bracket body>",
        "name": "<the clearance hole's mouth edge from query_geometry>"
      }
    }
  },

```

```
    "use": "axis"
  },
  "rotation": {
    "position": "0 deg",
    "limits": { "min": "0 deg", "max": "0 deg" }
  }
}
```

Appearance and the parts list

- `component.update` with `"appearance":{"finish":"black-oxide"}` colours a whole component over its part, for example one of two equal brackets.
- `items.reorder` orders the parts list. Parts lists come top level, indented or parts only; drawings of the assembly show them (see Drawings).

Assembly commands

`document.rename`, `parameter.create`, `component.insert`, `component.update`, `component.replacePart`, `component.delete`, `joint.create`, `joint.update`, `joint.delete`, `items.reorder`, `interference.markIntended`, `interference.unmark`, and for mechanisms `group.create`, `relation.create`, `study.create`, `exploded.create` and their updates. Read them with `describe_commands`.

Mechanisms

Mechanisms are assemblies whose parts move: linkages that close a loop, gears and belts that couple joints, screws that turn into a slide, and motion studies that play the motion and check it for collisions. Read Assemblies first: everything here is built from its joints.

Closed loops

A joint between two components the joints already place closes a loop (a four-bar linkage, a slider-crank) and is solved, not only checked: the loop's other axes follow.

- Mark the axis that drives the loop "driven": true (`{"rotation":{"position":"30 deg","driven":true}}`); `inspect_assembly` shows the values of the axes it moves with `solvedByLoop`.
- A loop that cannot close at that position is a `JOINT_CONFLICT` on its closing joint: check the link lengths, or the driven position against the loop's range.
- Ball, planar and pin-slot joints are the usual way to leave a loop the freedom it needs (a pin in a slot, a rod end).

Rigid groups

`group.create` makes components move as one, each where it is now relative to the first, without picking faces: `{"type":"group.create","id":"$g","components":[<component>,<component>]}`. A component is in one group at most. Use it for a weldment or parts that are bolted together and never move apart.

Relations

Relations couple two joints' axes: drive the first, the second follows, and the second's limits stop the first.

Type: Ratio

gear: Turns of b per turn of a; b turns the other way.

belt: Turns of b per turn of a, the same way.

rackPinion: How far b slides per turn of a, a length: " $\pi * 10 \text{ mm}$ " for a $\varnothing 10$ pinion.

screw: How far b slides per turn of a: the screw's lead.

```
{
  "type": "relation.create",
  "id": "$feed",
  "relation": {
    "type": "screw",
    "a": { "joint": "<lead screw's revolute joint>", "axis": "rotation" },
    "b": { "joint": "<carriage's slider joint>", "axis": "translation" },
    "ratio": "2 mm"
  }
}
```

Side b's own position is where it is while a is at zero. "reversed": true turns the direction around.

Motion studies

A motion study is keyframes of joint values over time. Axes move in straight lines between keyframes and hold outside them; playing a study changes nothing in the assembly.

```
{
  "summary": "A study that turns the shaft once in 4 s",
  "steps": [
    {
      "label": "Full turn",
      "commands": [
        {
          "type": "study.create",
          "id": "$spin",
          "study": {
            "name": "Full turn",
            "duration": 4,
            "keyframes": [
              {
                "time": 0,
                "values": [
                  {
                    "joint": "<Shaft on plate.$turn>",
                    "axis": "rotation",
                    "value": "0 deg"
                  }
                ]
              },
              {
                "time": 4,
                "values": [
                  {
                    "joint": "<Shaft on plate.$turn>",
                    "axis": "rotation",
                    "value": "360 deg"
                  }
                ]
              }
            ]
          }
        }
      ]
    }
  ]
}
```

Then `check_motion` with the study's id runs it and lists every pair of components that collide, from when to when and how much. It checks every degree or half millimeter of the fastest axis, an axis that turns a lead screw (side a of a screw relation) by half a millimeter of the travel it makes (a 30-turn Tr12x3 screw is 180 poses), or as many samples as you ask between from and to (seconds). A study's "sampling": "fine" (`study.update`) checks the screw by the degree too, for a handwheel or a cam on it that could hit something within a turn; when its time runs out it answers complete: false with `checkedTo`, so check the rest with `from` or fewer samples. Check the ends of travel and the poses the user cares about with `check_interference` and positions too.

The user plays studies and exports them as a video or a GIF.

Exploded views

An exploded view moves components apart in steps, along an axis of the assembly or a joint's axis, to show how it goes together; showing it changes nothing:

```

{
  "type": "exploded.create",
  "id": "$x",
  "view": {
    "name": "Service view",
    "trails": true,
    "steps": [
      {
        "components": ["<cover>"],
        "direction": { "axis": "z" },
        "distance": "30 mm"
      },
      {
        "components": ["<shaft>"],
        "direction": { "joint": "<shaft's joint>" },
        "flip": false,
        "distance": "40 mm"
      }
    ]
  }
}

```

Commands

group.create, group.update, group.delete, relation.create, relation.update, relation.delete, study.create, study.update, study.delete, exploded.create, exploded.update, exploded.delete. Read them with describe_commands.

Drawings

A drawing is a document of its own that draws a part or an assembly (its source, at the latest revision) on sheets: views, dimensions and annotations whose values come from the model. When the part changes, the drawing follows. On a drawing use `inspect_drawing` and `get_view`; the part tools answer only on parts.

Read Working with plans first.

Standard, sheets and template

- A drawing follows ISO (A4 to A0 sheets, first angle projection) or ASME (ANSI A to E sheets, third angle), and shows millimeters or inches (`inspect_drawing: standard, units`). Change them with `drawing.update`, and the sheets' size and projection with `sheet.update`, in one plan. Use ASME and inches when the user works in inches. Dimension values in `inspect_drawing` stay in mm whatever the drawing shows.
- Sheets are drawn on a template: the standard's own (ISO 7200 or ASME Y14.1, with zones) unless `drawing.update` names a template document of the project (`{ "kind": "document", "documentId": ... }`); `list_projects` lists each project's templates and `create_drawing` takes a `templateId`.
- Title block fields are typed per sheet with `sheet.update titleBlock`, by the template's field keys (title, number, material, checker, approver, ...); empty ones come from the part and the account. Revisions go in `drawing.update revisions` (`[{rev, description, date, by}]`, the whole list); the revision table lists them and the latest goes in the title block.
- `sheet.update` also sets scale ("1:2"), size, orientation, projection, `generalTolerance` ("ISO 2768-m") and `generalSurface`.

Views

- Orientations: front (looks along +Y, Z up), back, top (looks down, Y up), bottom, right, left, iso.
- A view's position is where the part's origin lands on the sheet, in mm from the lower left. Projected views name their parent and stay aligned with it; the sheet's projection method decides on which side they go.
- The `views.standard` helper places the standard views of the source, laid out by the projection method and scaled to fit:

```
{
  "summary": "Front, top, right and iso views of the plate",
  "steps": [
    {
      "label": "Views",
      "commands": [
        { "type": "views.standard", "views": ["front", "top", "right", "iso"] }
      ]
    }
  ]
}
```

- `view.create`, `view.update` (orientation, scale, hidden lines, tangent edges, label, breaks), `view.move` and `view.delete` change single views.
- Sections show what is inside: `views.section` cuts a front, top or side view along a line across the part (straight, half, or in steps for an offset section), hatches the cut faces and places the section beside

its parent with its letter (A-A). Use one where hidden lines would cross, or the inside matters (bores, pockets, tapped holes), and dimension inside features on the section, on the cut edges get_view lists for it.

- Details (views.detail) show small features larger; auxiliary views (views.auxiliary) show an inclined face at its true size; breaks (view.update breaks) shorten a long view. Dimension on them like on any view; values stay true.

Dimensions and annotations

Annotations reference edges by the topological names get_view returns for that view; their values come from the model, never from you.

```
{
  "type": "annotation.create",
  "id": "$width",
  "annotation": {
    "kind": "dimension",
    "view": "<front view>",
    "type": "horizontal",
    "refs": [
      { "kind": "edge", "name": "<left edge from get_view>", "at": "start" },
      { "kind": "edge", "name": "<right edge from get_view>", "at": "start" }
    ],
    "offset": [0, -12]
  }
}
```

- Dimension types: horizontal, vertical, aligned (one line edge gives its length, two edges the distance between them), radius and diameter (a circle seen face on), angular, chamfer, ordinate (from a datum along an axis), arcLength. offset places the dimension line in sheet mm from the middle of its edges; 10 to 15 mm away from the view reads well. "text": "4x <value>" adds words around the value. Dimensions with the same set move together (baseline or chain).
- Hole callouts: {"kind": "holeCallout", "view": "<view>", "ref": "<a circle of the hole seen end on>", "position": [x,y]} writes the hole as the part makes it (count, drill, depth, counterbore, countersink, thread: 4x M4 - 6H ? 8). A through hole has no depth: ISO writes only its diameter (2x Ø5.5), ASME adds THRU. Use one on every hole and tapped hole instead of a plain diameter.
- Tolerances on a size: {"type": "symmetric", "value": 0.1}, {"type": "deviation", "upper": 0.1, "lower": -0.05}, {"type": "limits", ...}, {"type": "fit", "hole": "H7"} or "shaft": "g6" (ISO 286, deviations from the tables), {"type": "basic"} for GD&T, {"type": "reference"}.
- GD&T: datums ({"kind": "datum", "letter": "A", ...} on an edge seen edge on, or on a dimension) and feature control frames ({"kind": "featureControlFrame", "characteristic": "perpendicularity", "zone": {"value": 0.02}, "datums": [{"letter": "A"}], ...}). Form tolerances take no datum; orientation and location need one; M and L only on a dimension. Broken rules show as GDT_INVALID.
- Surface texture: {"kind": "surfaceSymbol", "ref": "<edge of the surface>", "process": "removal", "parameters": ["Ra 1.6"], ...}, and generalSurface on the sheet for every other surface.
- Center marks, centerlines and notes: centerMark, centerline, note.
- A dimension whose edge is gone is dangling and shows no value: reattach it with annotation.update (new refs), or delete it.

Dimensioning a part completely

Unless the user asks for less, dimension a drawing so someone can make the part from it without measuring the model:

1. The overall size in all three directions.
2. Every hole by a hole callout, and its position from the part's edges or from a datum.
3. Angles by angular dimensions, chamfers by chamfer dimensions, fillet radii.
4. Every boss, slot and step; patterns with count and spacing (4x <value>).
5. A fit on every size that mates with another part (a bore for a shaft or a bearing H7, a shaft g6, or as the user says), and the general tolerances in the title block (ISO 2768-m unless asked).
6. A datum on the face or feature a fit is measured from, and a frame (perpendicularity or position) tying the fit to it; surface texture on fitted faces and a general symbol for the rest.
7. A section view where hidden lines would cross inside the part.

Each size once, in the view that shows its shape best. Before you answer, check `inspect_drawing`:

- `missingSizes`, `missingCallouts`, `unfittedBores`, `crowded` and `gdtProblems` must be empty. `missingSizes` lists overall and circle sizes no dimension shows yet; it cannot see positions, so check those yourself. `crowded` lists texts a line or another text runs through: move those annotations clear of the views and of each other.
- No dimension may be dangling: reattach or delete one you leave behind, and say so.

Drawings of assemblies

Refs also name the component (`{"kind":"edge","name":...,"component":...}`). A parts list is `{"kind":"partsList","sheet":<sheet>,"source":<assembly source id>}` (above the title block, item 1 at the bottom), and balloons `{"kind":"balloon","view":<view>,"component":<component>,"position":[x,y]}` point at components.

Drawing commands

`document.rename`, `drawing.update`, `sheet.update`, `view.create`, `view.update`, `view.move`, `view.delete`, `annotation.create`, `annotation.update`, `annotation.delete`, and the helpers `views.standard`, `views.section`, `views.detail`, `views.auxiliary`. Read them with `describe_commands`.

Project parameters

Every project has one parameter table: sizes that several of its documents must agree on (a spindle's center height, rail spacing, a bolt circle, a sheet thickness) are set there once, and parts and assemblies read them as project.<name>. When the table changes, every document that reads it follows, without a new revision of its own.

Read Working with plans first.

When to use it

- Put a size in the table when two documents must agree on it; never repeat a number another document owns. A size only one part uses stays a parameter of that part.
- A bought part's length in one assembly is not a project parameter: set it on the component (see Assemblies, component parameters).

The table

list_projects and create_project give each project's parametersId: the table is a document like any other. On it, inspect_document lists the parameters with their values, kinds, groups and usedBy (the documents that read each, also through other parameters of the table, through, and assemblies and drawings by the documents they use, via), and preview_plan changes it:

```
{
  "summary": "Machine sizes the headstock, tool post and bed share",
  "steps": [
    {
      "label": "Machine",
      "commands": [
        {
          "type": "parameter.create",
          "name": "railHeight",
          "expression": "15 mm",
          "group": "Machine"
        },
        {
          "type": "parameter.create",
          "name": "centerHeight",
          "expression": "80 mm",
          "group": "Machine"
        },
        {
          "type": "parameter.create",
          "name": "axisZ",
          "expression": "railHeight + centerHeight",
          "group": "Machine",
          "comment": "The spindle's axis above the bed"
        }
      ]
    }
  ]
}
```

- Expressions in the table use the table's own names and units, never project. and never a document's parameters.
- parameter.update changes an expression, comment, group or aliases by id; parameter.delete is refused while another parameter or a document reads the name; parameter.move orders them.

- Renaming a name documents read is refused for now: add a parameter with the new name, move the documents to it, then delete the old one.

Reading it

In a part or an assembly, any expression can read the table:

`{"type":"parameter.create","name":"axisH","expression":"project.axisZ - 10 mm"}`, a feature value

`"project.wall * 2"`, a joint offset, a configuration cell, `{project.partNumber}` in sketch text.

`inspect_document` on a part lists the table's names under `project`, with the values and whether this document uses each; a parameter that reads the project says so (`readsProject`).

Changing it

A change of the table reaches every document that reads a changed name. `preview_plan` on the table rebuilds them with the new values before anything is saved and lists them in `affected`: each document with `ok`, the failing feature or joint and its message, and an assembly's degrees of freedom. The plan applies at once when they all rebuild (unless the user approves changes); when one fails it waits for `Apply`, and the user decides. Tell the user which documents the change reached or would break.

A library version keeps the values it was published with; changing the table does not change parts inserted from the library.

Project notes

A project's notes are Markdown documents next to its parts: what the product is for, what was decided and why, what was checked and what is still open. The user and every agent after you read them, so a note is where a session's understanding outlives its chat. Notes change only through plans, like any document.

Read Working with plans first.

Read them first

Before you work on a project, read its notes: `list_projects` lists each project's notes (`noteId`), and `inspect_document` on a note returns its text, its outline (the headings) and the documents it links to. Follow what they say was decided unless the user asks otherwise, and do not redo checks they list for the revision that is still current.

Keep them at the end

At the end of a session that decided, built or checked something, propose an update to the project's notes. Make a note with `create_note` when the project has none: it starts with these headings, or only the ones you pass in sections (for example `["Decisions", "Checked"]`).

Heading: What goes there

Purpose and requirements: What it is for, what it must do, its limits (loads, travel, sizes it must fit)

Decisions: What was chosen, the alternatives left and why: a bronze nut, not a printed one, because it wears

Parameters: What the main parameters mean and their safe ranges, by name (`railHeight`); never copy values

Bought parts: Standard and library parts, materials, where they come from

Making it: Print or machining settings, tolerances and fits

Checked: Interference, motion studies, mass: what passed, at which revision of which document

Open questions: What is not decided or not done yet

Write short bullets of facts, with the revision a check was made at. Link a document with an ordinary Markdown link whose address is `moket:/d/<documentId>`: the preview shows the document's current name, so a rename breaks nothing. Never put secrets, tokens or personal data in a note.

Changing a note

- `note.editSection` changes one section: mode `append` adds to the end of the section under heading (and adds the heading at the end of the note when it has none), `replace` replaces the section's text and keeps its heading. A heading matches by its text at any level, without case.
- `note.setText` replaces the whole text: use it only for a new note or when the user asks to rewrite one, and keep what was there.
- `document.rename` renames the note.

The plan's changes list the sections it adds, changes and removes, with the lines added and removed, and its sections give each one's text before and after. A plan applies at once unless the user approves changes: show the user what you wrote from sections, and to let them read it before it goes in, ask them to turn on Settings > Mocket AI > Ask before applying AI changes with Any change checked

(Changing what is in the part covers changed sections, not new ones). A note has no geometry, so nothing rebuilds.

```
{
  "summary": "Keep what this session decided and checked",
  "steps": [
    {
      "label": "Decisions and checks",
      "commands": [
        {
          "type": "note.editSection",
          "heading": "Decisions",
          "text": "- Cross feed: Tr12x3 screw in a bronze nut (CuSn8): a printed nut wears out",
          "mode": "append"
        },
        {
          "type": "note.editSection",
          "heading": "Checked",
          "text": "- Interference of the lathe: none, 226 pairs (rev 30)\n- Motion study Cross feed: no collisions",
          "mode": "replace"
        }
      ]
    }
  ]
}
```

Library and standard parts

Some parts of a machine are designed; most of the rest is bought: linear rails and carriages, motors, couplings, lead screws, pulleys and belts, sensors, profiles, screws, nuts, washers and bearings. Mocket keeps the bought ones in two places, so they are never modeled twice.

Read Assemblies first for inserting components.

Standard parts

Screws, nuts, washers, ball bearings and O-rings of ISO and ANSI standards are built in.

`find_standard_part` (`{"query":"ISO 4762 M4"}` or `"608 bearing"`) works on any document and gives each part's id, standard, units, material, where its origin is, its parameters, and the rows that match with their configuration ids and dimensions. Insert one with `component.insert`
`{"part":<id>,"configuration":<row id>}`; standard parts are never copied into the project. How to fasten a hole with them is in Assemblies.

The component library

The user's component library holds bought parts someone modeled once, by maker, series and part number, in versions.

1. Before modeling a bought part, call `find_library_part` with its maker, series, part number or size. It lists matching library items with their versions, and the matching standard parts too.
2. Insert a library item with `component.insert` `{"part":<version.part>,"configuration":<a size's configuration>}`. The component keeps that version: `inspect_assembly` shows `part.library {item, version}` on library components, and `component.replacePart` moves components of an older version to the latest one `find_library_part` returns.
3. Set what changes per use on the component (a rail's length): `component.update` parameters. A library version keeps the project values it was published with.

Modeling a bought part

When the library has no such component, model it as a part of the project from the maker's datasheet:

- The outline, the mounting holes and the faces other parts attach to, at the datasheet's dimensions. Ask the user for the datasheet or its link when you do not have it; never guess the size of a real product.
- Its sizes as configurations, one row per size, named as the maker names them (see Parts, configurations), and the maker's part number in its name.
- Put its origin where it mounts (a rail's bottom face at its first hole, a motor's flange at its shaft axis), so joints are easy.
- Detail nothing attaches to can stay simple.

Then offer to publish it with `publish_to_library` (maker, series, part number, the datasheet link, a note on where the sizes come from), so the next project inserts it instead of modeling it again. Publish only after the user agreed; a later change of the same part becomes the item's next version.

Leather

The Leather Workbench is Mocket's own plugin for leather goods: wallets, card holders, straps and cases. Its features work like any feature of a part: you add them with `feature.create` in a plan and read them with `describe_commands` first. A leather part is flat pieces stacked on each other, stitch lines through them, and folds that close it.

Read `Parts` first. `inspect_document` lists the user's plugins under `plugins` with their feature types; Leather is there unless the user turned it off in `Settings > Plugins`.

Features

Feature: What it does

`io.mocket.leather.piece`: A flat piece from one closed region of a sketch, thickness `thick`, with its leather, finish, grain, name and count.

`io.mocket.leather.stitchPath`: A stitch line: holes along edges of a piece's face, margin in from them, at a pitch, spread by arc length.

`io.mocket.leather.stitchHoles`: Punches a stitch line's holes through every piece in bodies, the same holes in each layer: chisel slits or round.

`io.mocket.leather.seamAllowance`: Moves a piece's cut line out from chosen edges for the pattern; the piece itself does not change.

`io.mocket.leather.fold`: Folds pieces along a sketch line by an angle with an inside radius; stacked layers fold together.

- A piece grows from its sketch along the sketch's normal; "flip": true grows it the other way. Its faces are named like an extrude's: `f(<piece>:start)` on the sketch, `f(<piece>:end)` on top, `f(<piece>:side:<line>)` round it.
- Stack a piece on another by sketching it on the lower one's top face (`"plane":{"kind":"face","ref":{"kind":"face","name":"f($outer:end)"}}`). On a face that faces up, the sketch's `x` and `y` are the part's `X` and `Y`.
- Leathers are veg-tan (outers, 1.2 to 2 mm), chrome-tan (pockets, 0.6 to 1 mm) and lining (0.4 to 0.8 mm); finishes natural, tan, brown, dark-brown, black, burgundy, navy, green. Use the user's leather and thickness.
- The usual pitch is 3.85 mm (a 4 mm iron), 3 mm for small goods; the margin 3 to 4 mm. corners hole puts a hole at every corner and evens the pitch between corners.

Stitching through layers

One stitch line and one `Stitch holes` through every layer it holds give every layer the same holes. Draw the stitch line on the top layer's face and list the pieces it goes through in bodies, bottom to top: a hole goes only where a piece is, so a pocket that covers part of the outer gets the holes over it.

Edges of a stitch line follow each other end to end. Name them from the piece and the lines of its sketch: in the same plan, a template's lines are `$outline.lines.bottom` (a rectangle's bottom, right, top, left; a profile's `$pocket.lines.0` from its first corner on).

```
{
  "summary": "Card holder 100 x 65 with a front pocket, stitched round",
  "steps": [
    {
```

```

"label": "Name and parameters",
"commands": [
  { "type": "document.rename", "name": "Card holder" },
  {
    "type": "parameter.create",
    "id": "$width",
    "name": "width",
    "expression": "100 mm"
  },
  {
    "type": "parameter.create",
    "id": "$height",
    "name": "height",
    "expression": "65 mm"
  },
  {
    "type": "parameter.create",
    "id": "$pocketHeight",
    "name": "pocketHeight",
    "expression": "45 mm"
  }
]
},
{
"label": "Outer, 1.4 mm veg-tan",
"commands": [
  {
    "type": "sketch.template",
    "id": "$outline",
    "name": "Outer",
    "template": {
      "kind": "rectangle",
      "width": "width",
      "depth": "height"
    }
  },
  {
    "type": "feature.create",
    "id": "$outer",
    "featureType": "io.moket.leather.piece",
    "name": "Outer",
    "params": {
      "profile": { "sketch": "$outline" },
      "body": "$outerBody",
      "thickness": "1.4 mm",
      "leather": "veg-tan",
      "finish": "brown"
    }
  }
]
},
{
"label": "Pocket on the outer, 0.8 mm chrome-tan",
"commands": [
  {
    "type": "sketch.template",
    "id": "$pocketOutline",
    "name": "Pocket",
    "template": {
      "kind": "profile",
      "corners": [
        { "x": "-width / 2", "y": "-height / 2" },
        { "x": "width / 2", "y": "-height / 2" },
        { "x": "width / 2", "y": "pocketHeight - height / 2" },
        { "x": "-width / 2", "y": "pocketHeight - height / 2" }
      ],
      "plane": {
        "kind": "face",
        "ref": { "kind": "face", "name": "f($outer:end)" }
      }
    }
  }
]
}

```

```

    }
  },
  {
    "type": "feature.create",
    "id": "$pocket",
    "featureType": "io.moket.leather.piece",
    "name": "Pocket",
    "params": {
      "profile": { "sketch": "$pocketOutline" },
      "body": "$pocketBody",
      "thickness": "0.8 mm",
      "leather": "chrome-tan",
      "finish": "brown"
    }
  }
]
},
{
  "label": "Stitch round the outer, through the pocket",
  "commands": [
    {
      "type": "feature.create",
      "id": "$stitch",
      "featureType": "io.moket.leather.stitchPath",
      "name": "Edge stitch",
      "params": {
        "edges": [
          {
            "kind": "edge",
            "name": "e(f($outer:end)|f($outer:side:$outline.lines.bottom))"
          },
          {
            "kind": "edge",
            "name": "e(f($outer:end)|f($outer:side:$outline.lines.right))"
          },
          {
            "kind": "edge",
            "name": "e(f($outer:end)|f($outer:side:$outline.lines.top))"
          },
          {
            "kind": "edge",
            "name": "e(f($outer:end)|f($outer:side:$outline.lines.left))"
          }
        ],
        "margin": "4 mm",
        "pitch": "3.85 mm"
      }
    },
    {
      "type": "feature.create",
      "id": "$holes",
      "featureType": "io.moket.leather.stitchHoles",
      "name": "Edge holes",
      "params": {
        "line": "$stitch",
        "bodies": ["$outerBody", "$pocketBody"],
        "shape": "chisel"
      }
    }
  ]
}
]
}
}

```

Checking stitch counts

A stitch line and Stitch holes report what they made: the preview's reports ("78 holes, pitch 3.8 to 3.83 mm, 298 mm of stitching"), the feature's report in `inspect_document` and its reports in `get_feature`,

with the count in details (holes). Two edges sewn to each other (the sides of a gusset and the panel it joins, a fold's two halves stitched closed) need the same number of holes: compare the reports, and change the pitch or the gaps (startGap, endGap on an open line) of one until they match. Tell the user the counts.

Folds

A fold turns the pieces in bodies about a line of a sketch on them (line:

{ "sketch": <sketch>, "curves": [<line>] }, by angle (180 deg folds them shut) with an inside radius. The side to the left of the line, seen from the side the sketch faces and walking from its start to its end, moves; otherSide moves the other one, and under folds away from the side the sketch faces. Every layer folds with its own radius, the stitches and the pieces lying wholly on the moving side ride along.

```
{
  "summary": "Key fob strap 120 x 25, folded in half",
  "steps": [
    {
      "label": "Strap and crease",
      "commands": [
        { "type": "document.rename", "name": "Key fob" },
        {
          "type": "sketch.template",
          "id": "$outline",
          "name": "Strap",
          "template": {
            "kind": "rectangle",
            "width": "120 mm",
            "depth": "25 mm"
          }
        }
      ],
    },
    {
      "type": "feature.create",
      "id": "$strap",
      "featureType": "io.moket.leather.piece",
      "name": "Strap",
      "params": {
        "profile": { "sketch": "$outline" },
        "body": "$strapBody",
        "thickness": "1.4 mm"
      }
    },
    {
      "type": "feature.create",
      "id": "$crease",
      "featureType": "core.sketch",
      "name": "Crease",
      "params": {
        "plane": {
          "kind": "face",
          "ref": { "kind": "face", "name": "f($strap:end)" }
        },
        "entities": [
          { "id": "$a", "kind": "point", "x": 0, "y": -20, "fixed": true },
          { "id": "$b", "kind": "point", "x": 0, "y": 20, "fixed": true },
          {
            "id": "$creaseLine",
            "kind": "line",
            "start": "$a",
            "end": "$b"
          }
        ],
        "constraints": []
      }
    }
  ]
}
```

```

    },
    {
      "label": "Fold the left half over",
      "commands": [
        {
          "type": "feature.create",
          "id": "$fold",
          "featureType": "io.moket.leather.fold",
          "name": "Fold",
          "params": {
            "line": { "sketch": "$crease", "curves": ["$creaseLine"] },
            "bodies": ["$strapBody"],
            "angle": "180 deg",
            "radius": "1 mm"
          }
        }
      ]
    }
  ]
}

```

Draw the crease as a core.sketch on the top layer's face with two fixed points and a line between them; placeholders work as entity ids. The line here runs up the middle, so the left half folds over onto the right. Fold last, after every stitch, hole and allowance: the pattern shows each piece flat as cut. A fold whose pieces need more room warns The pieces inside the fold need an inside radius of at least ... mm: raise the radius to it, never leave the warning. Folds are geometric: the leather does not stretch or thin.

A wallet

A bifold wallet is an outer, a liner stacked on it, and pockets stacked on the liner on each side of the fold:

1. Parameters for the closed size, thicknesses, margin and pitch; the outer and the liner the open size (twice the closed width).
2. Pockets on the liner, each stack on the one under it, each pocket a little lower than the one it covers so the cards show.
3. A stitch line round each pocket stack, through the stack and the liner and outer under it; a stitch line round the liner through the liner and the outer. Compare the counts of seams that must match.
4. The fold on the liner's top face at the middle, through every piece, 180 deg, with the radius it asks for.

The pattern and plugins of the user

- The user exports the pattern (the Leather workbench's Export pattern: tiled PDF, PDF, SVG and DXF at 1:1). You cannot; name the pieces (name) and their count so the pattern reads well.
- A user's own plugin runs only in a browser tab: on Mocket's server its features are described, and adding or changing one answers PLUGIN_NEEDS_TAB. Ask the user to open Mocket in a tab, then call again.
- A plugin the user does not have on shows missing in inspect_document: its features keep their last result, frozen, and cannot be changed.

Working over MCP

You reach Mocket from outside, as a client of its MCP server: Claude Code, Claude, ChatGPT, Codex or another agent. No Mocket tab or editor needs to be open for you, and you never ask the user to open a document or to make a project. You use the same tools as Mocket AI in the editor, and your changes follow the user's approvals.

Projects and documents

1. Start with `list_projects`: the user's projects with their documents, templates, notes (`noteId`, read them first: see Project notes) and parameter table (`parametersId`).
2. `list_agent_documents` with a `projectId` lists a project's documents; without one, what is open now.
3. `create_project` makes a project; `create_part`, `create_drawing` (from a part or an assembly, with a template), `create_assembly` and `create_note` make documents. Give the user the link `create_part` returns when they want to look.
4. Work on any document by its `documentId`: Mocket opens it for you in the background, and the user follows your work in Agent activity. Every applied plan is a version of the document in your name.

Plans that wait

A plan the user must approve waits in Agent activity and in the document's editor. `commit_plan` with its `previewId` answers pending until the user decides, then applied or discarded: call it again to keep waiting, and tell the user what waits for them. A plan that applied at once (`autoApplied`) needs no `commit_plan`.

When something is unavailable

- `CAD_SESSION_UNAVAILABLE` means nothing can open documents for you right now: ask the user to open Mocket in a browser tab, or to turn on Work without an open tab in Settings > Agents.
- `HOST_UPDATING` means Mocket is restarting on a new version: call again in a few seconds.
- `HOST_OUTDATED` means Mocket was updated and the user's tab that runs your tools still has the old version open in an editor: ask the user to reload that tab.
- `EDITOR_NOT_READY` means the document is opening in the user's editor tab: call again in a few seconds.
- `DOCUMENT_CONFLICT` means the user's editor holds changes made on an older revision: ask the user to click Choose what to keep in the banner over the view in that tab and pick Keep my changes or Use rev N (the newest revision: usually right when they did not mean to change anything there), then call again.
- revision in what the tools return is the document's revision in the cloud, the one `list_agent_documents` lists and the user sees in Versions.
- A tool or a guide name that a guide mentions is missing from your tool list: Mocket was updated after you connected, and your client keeps the list it read then. Ask the user to reconnect Mocket in the client (`/mcp` in Claude Code).
- A call that fails without an answer may still have applied: inspect the document and compare its revision before you send the plan again.

Guides and this skill

`read_guide` returns any guide by name (working, parts, sketches, assemblies, mechanisms, drawings, project-parameters, notes, library, mcp), and without a name lists them. They are also MCP resources (`moket://guides/<name>`). The text Mocket answers is always current; a skill installed earlier is a copy of the day it was downloaded.

Feedback

When a tool behaves differently from its description, when you need something Mocket cannot do, or when a description misled you, tell Mocket's team once with `send_feedback`, then go on with the task.