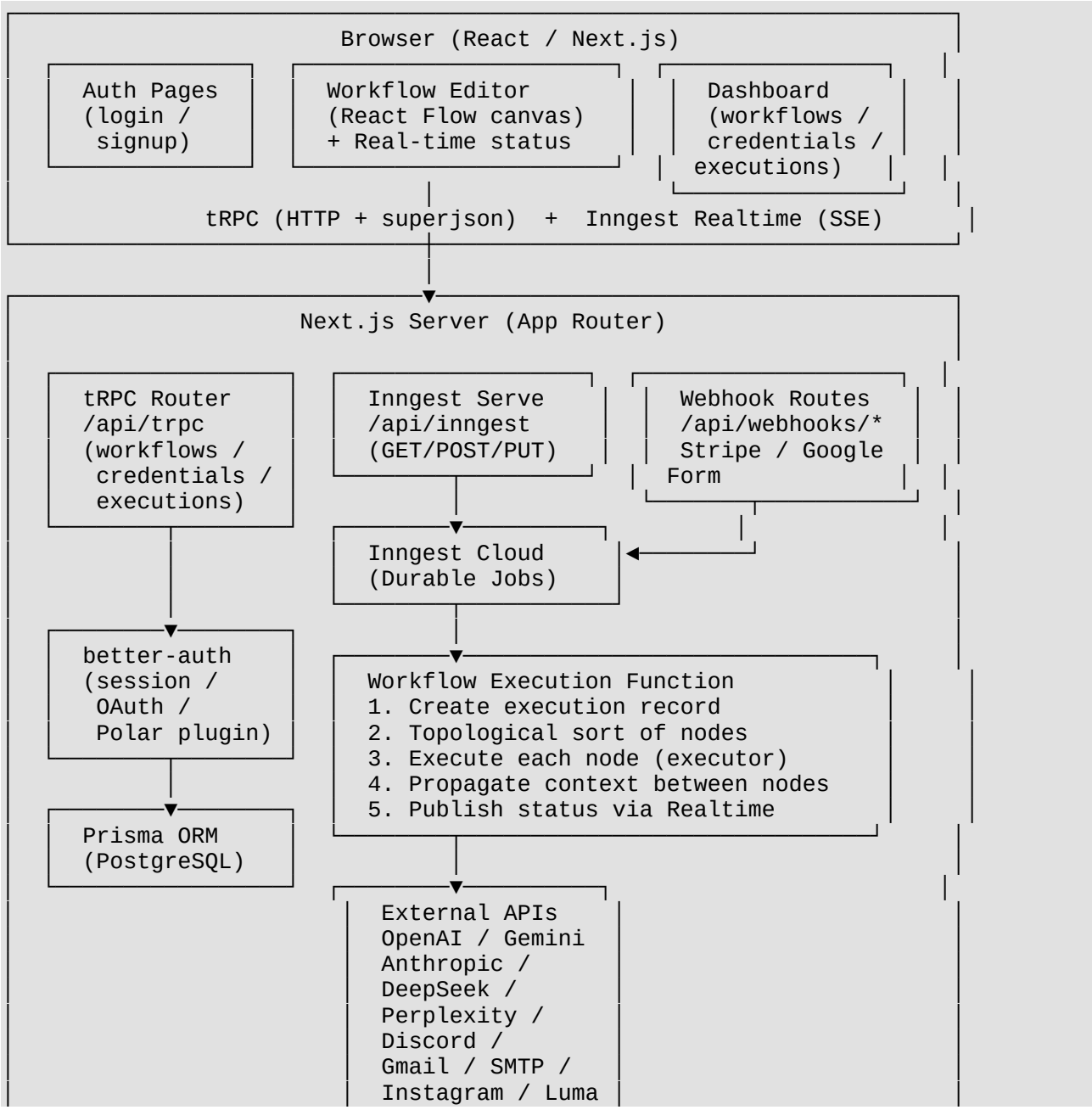
ConnectFlow — Architecture & Technical Documentation

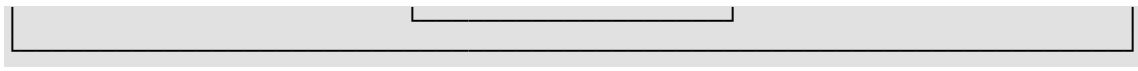
> **ConnectFlow** is a full-stack, no-code workflow automation platform inspired by n8n and Zapier. Users build visual pipelines by connecting trigger nodes to action nodes on a drag-and-drop canvas. When a trigger fires, an Inngest background-job fan-out executes every node in topological order, streaming real-time status updates back to the browser over a persistent WebSocket-style subscription.

Table of Contents

---

1. High-Level Architecture





---

2. Project Structure

```
connectflow/
├── prisma/
│   └── schema.prisma              # Full database schema
│
├── src/
│   ├── app/                       # Next.js App Router
│   │   ├── (auth)/                # Unauthenticated route group
│   │   │   ├── login/page.tsx
│   │   │   └── signup/page.tsx
│   │   ├── (dashboard)/           # Authenticated route group
│   │   │   ├── (rest)/            # Standard dashboard pages
│   │   │   │   ├── workflows/
│   │   │   │   ├── credentials/
│   │   │   │   └── executions/
│   │   │   └── (editor)/          # Full-screen editor layout
│   │   │       └── workflows/[workflowId]/
│   │   └── api/
│   │       ├── auth/[...all]/     # better-auth catch-all route
│   │       ├── inngest/           # Inngest function serve endpoint
│   │       ├── trpc/[trpc]/       # tRPC HTTP handler
│   │       ├── trigger/           # Programmatic API trigger endpoint
│   │       └── webhooks/
│   │           ├── stripe/        # Stripe event ingestion
│   │           └── google-form/   # Google Form submission ingestion
│   │
│   ├── features/                  # Vertical feature slices
│   │   ├── auth/
│   │   ├── credentials/
│   │   ├── editor/
│   │   ├── executions/
│   │   │   ├── components/        # One sub-folder per node type
│   │   │   │   ├── anthropic/     # node.tsx, dialog.tsx, action.ts,
executor.ts
│   │   │   │   ├── discord/
│   │   │   │   ├── gemini/
│   │   │   │   ├── gmail/
│   │   │   │   ├── http-request/
│   │   │   │   ├── instagram/
│   │   │   │   ├── luma/
│   │   │   │   ├── openai/
│   │   │   │   ├── deepseek/
│   │   │   │   ├── perplexity/
│   │   │   │   └── custom_mail/
│   │   │   ├── hooks/
│   │   │   ├── lib/
│   │   │   │   └── executor-registry.ts
│   │   │   ├── server/
│   │   │   │   └── router.ts      # tRPC executions router
│   │   │   └── types.ts
│   │   ├── subscriptions/
│   │   ├── triggers/
│   │   │   └── components/        # One sub-folder per trigger type
│   │   │       ├── api-trigger/
│   │   │       ├── google-form-trigger/
│   │   │       ├── manual-trigger/
│   │   │       └── stripe-trigger/
```

```
│   │   └── workflows/
│   │
│   ├── inngest/
│   │   ├── channels/              # Inngest Realtime channel definitions
│   │   ├── workflows/             # Special workflow trigger helpers
│   │   ├── client.ts              # Inngest client singleton
│   │   ├── functions.ts           # Main executeWorkflow function
│   │   └── utils.ts               # topologicalSort, sendWorkflowExecution
│   │
│   ├── lib/
│   │   ├── api-key.ts             # AES-256-CBC key generation/decryption
│   │   ├── auth.ts                # better-auth server config
│   │   ├── auth-client.ts         # better-auth browser client
│   │   ├── auth-utils.ts          # requireAuth / requireUnauth helpers
│   │   ├── db.ts                  # Prisma singleton (global cache)
│   │   └── polar.ts               # Polar SDK client
│   │
│   ├── trpc/
│   │   ├── init.ts                # Procedures: base / protected / premium
│   │   ├── client.tsx             # TRPCReactProvider + QueryClient
│   │   ├── server.tsx             # Server-side caller factory
│   │   ├── query-client.ts        # TanStack QueryClient factory
│   │   └── routers/_app.ts        # Root AppRouter
│   │
│   └── config/
│       ├── constants.ts           # Pagination defaults
│       └── node-component.ts      # NodeType → React component registry
```

---

3. Technology Stack

Layer	Technology	Version	Purpose
Framework	**Next.js**	16	Full-stack React, App Router, Server Components, Turbopack
Language	**TypeScript**	5	End-to-end type safety
UI Components	**shadcn/ui** + **Radix UI**	latest	Accessible, headless component primitives
Styling	**Tailwind CSS**	4	Utility-first CSS with JIT
Canvas	**React Flow (@xyflow/react)**	12.x	Drag-and-drop node-graph editor
Data Fetching	**tRPC**	11	Type-safe RPC without schema boilerplate
Server State	**TanStack Query**	5	Caching, mutation state, optimistic updates
Client State	**Jotai**	2	Atomic, minimal global state (editor instance)
Forms	**React Hook Form** + **Zod**	7 / 4	Strongly-typed form validation

Layer	Technology	Version	Purpose
Auth	**better-auth**	1.3	Session, OAuth (GitHub, Google), Polar plugin
Database	**PostgreSQL** via **Prisma**	6	Relational store, type-safe ORM, migrations
Background Jobs	**Inngest**	3.x	Durable functions, retries, step-level checkpointing
Realtime	**@inngest/realtime**	0.4	Server-sent status updates streamed to the browser
AI Models	**Vercel AI SDK**	5	Unified interface to Anthropic, OpenAI, Google, DeepSeek, Perplexity, Luma
Email	**Nodemailer**	7	SMTP sending for Gmail and custom SMTP
HTTP Client	**ky**	1	Lightweight fetch wrapper for webhook/API calls
Templates	**Handlebars**	4.7	Dynamic variable interpolation inside node config
Subscriptions	**Polar** + **@polar-sh/better-auth**	0.40 / 1.3	Checkout, customer portal, subscription gating
Monitoring	**Sentry**	10	Error tracking & performance monitoring
Linting	**Biome**	2.2	Fast unified linter + formatter
IDs	**@paralleldrive/cuid2**	3	Collision-resistant unique IDs for Inngest events
Slug Generation	**random-word-slugs**	0.1	Human-readable workflow names on creation
URL State	**nuqs**	2	Type-safe URL search params

---

4. Frontend Architecture

4.1 Next.js App Router

ConnectFlow uses the **Next.js App Router** (introduced in Next 13, now the default). This gives:

- **React Server Components (RSC)** by default — zero client JS for purely-informational pages
- **Server Actions** used in `"use server"` files such as `action.ts` inside each node folder
- **Streaming + Suspense** for data-prefetching; Suspense boundaries in layout files allow incremental page rendering
- **Route Groups** `(auth)` and `(dashboard)` isolate layouts without affecting the URL

The build command uses `--turbopack` for both `dev` and `build`, enabling significantly faster HMR and build times over webpack.

4.2 Routing & Layouts

```
/                     → redirect to /workflows
/(auth)/login         → unauthenticated layout (centered card)
/(auth)/signup        → same
/(dashboard)/         → sidebar + header layout (AppSidebar, AppHeader)
  (rest)/workflows    → workflow list page
  (rest)/credentials  → credentials CRUD page
  (rest)/executions   → execution history list
  (editor)/workflows/[workflowId]  → full-screen editor (no sidebar)
```

Authentication guard is enforced at the layout level:

```
// src/app/(dashboard)/layout.tsx
const session = await requireAuth(); // redirects to /login if no session
```

`requireAuth` calls `auth.api.getSession({ headers })` server-side and issues a `redirect("/login")` if the session is absent. Similarly, `requireUnauth` guards the auth pages to redirect logged-in users to `/`.

4.3 Visual Workflow Editor (React Flow)

The editor is built on **`@xyflow/react`** (React Flow v12), a battle-tested graph editor library.

Key concepts:

React Flow Concept	ConnectFlow Usage
`Node`	Each workflow step (trigger or action)
`Edge`	A directed connection from one node's output to another's input
`NodeTypes`	Map of `NodeType` enum → React component (`src/config/node-component.ts`)
`ReactFlowInstance`	Stored in Jotai atom; extracted by the Save button to read current canvas state

Node Component structure — every node type follows the same 4-file pattern:

```
src/features/executions/components/<type>/
├── node.tsx      → React Flow node component (canvas widget)
├── dialog.tsx    → Settings dialog (React Hook Form + Zod schema)
├── action.ts     → "use server" — Inngest Realtime token fetcher
└── executor.ts   → Server-side execution logic (runs inside Inngest)
```

Saving a workflow: The `EditorSaveButton` reads nodes and edges from the `ReactFlowInstance` atom and fires a `trpc.workflows.update` mutation. The server performs a database transaction that:

1. Deletes all existing `Node` rows for the workflow (cascade deletes `Connection` rows via FK)
2. Bulk-inserts new `Node` rows with current positions and data
3. Bulk-inserts new `Connection` rows from edge definitions
4. Touches `workflow.updatedAt`

Node configuration is persisted inside the `data: Json` column of the `Node` model. Each node type has its own typed schema; the dialog form writes values back into this JSON via `setNodes` in the React Flow state.

4.4 State Management (Jotai + TanStack Query)

ConnectFlow uses a **two-tier state model**:

Jotai (client-side atomic state) — used exclusively for editor-level global state:

```
// src/features/editor/store/atoms.ts
export const editorAtom = atom<ReactFlowInstance | null>(null);
```

The `ReactFlowInstance` is stored so that the Save and Execute buttons, which live outside the `<ReactFlow>` tree, can read the current canvas state without prop-drilling.

TanStack Query (server state) — all data fetched from the server is managed here:

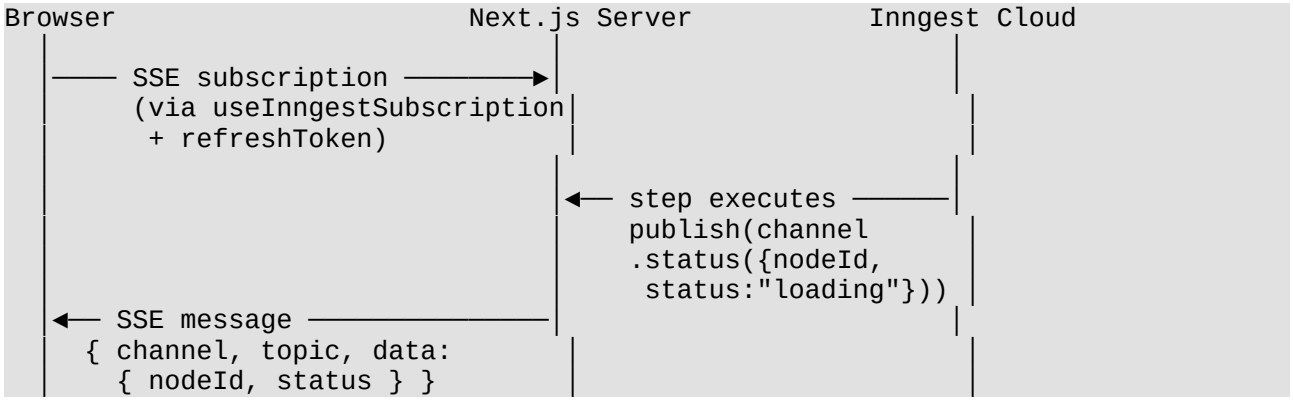
- `useWorkflows`, `useSuspenseWorkflow` → workflow list and individual workflow
- `useCredentials`, `useCredentialsByType` → credential list filtering by type
- `useExecution`, `useExecutions` → execution history
- `useHasActiveSubscription` → Polar subscription status with 5-second polling when inactive

The tRPC client is configured with `superjson` as the transformer, enabling Date, Map, Set, and BigInt values to pass through the wire without manual serialisation.

4.5 Real-Time Node Status (Inngest Realtime)

One of ConnectFlow's most technically sophisticated features is live per-node execution feedback streamed to the browser while a workflow runs.

Architecture:



Channel definition (one per node type):

```
// src/inngest/channels/anthropic.ts
export const anthropicChannel = channel("anthropic-execution")
  .addTopic(
    topic("status").type<{
      nodeId: string;
      status: "loading" | "success" | "error";
    }>()
  );
```

Client subscription — the `useNodeStatus` hook:

```
const { data } = useInngestSubscription({ refreshToken, enabled: true });

useEffect(() => {
  const lastMessage = data
    .filter(msg => msg.channel === channel && msg.data?.nodeId === nodeId)
    .sort(byCreatedAt)[0];

  if (lastMessage?.kind === "data") setStatus(lastMessage.data.status);
}, [data, nodeId, channel, topic]);
```

The `refreshToken` is a server action that calls `getSubscriptionToken(inngest, { channel, topics })`. Tokens are short-lived and the hook automatically rotates them.

Server-side publishing happens inside each executor:

```
await publish(anthropicChannel().status({ nodeId, status: "loading" }));
// ... do AI call ...
await publish(anthropicChannel().status({ nodeId, status: "success" }));
```

The `publish` function is injected by Inngest into the workflow handler and is available to every executor through the `NodeExecutorParams` interface.

4.6 UI Component System (shadcn/ui)

ConnectFlow uses **shadcn/ui** — a curated collection of copy-paste components built on **Radix UI** primitives and styled with Tailwind CSS.

All components live in `src/components/ui/` and are fully customisable:

- `Sidebar` with collapsible icon mode
- `Dialog` / `Sheet` for node configuration panels

- `Form` + `FormField` wrappers around React Hook Form
- `Select`, `Input`, `Textarea`, `Button` for form controls
- `Card`, `Badge`, `Separator` for layout
- `Sonner` for toast notifications
- `Breadcrumb` for editor navigation

---

5. Backend Architecture

5.1 tRPC API Layer

ConnectFlow's entire API surface is defined with **tRPC v11**, giving complete end-to-end type safety from database query to React component without generating any API client code.

Procedure tiers:

```
baseProcedure          // No auth required (currently unused in routes)
  └── protectedProcedure   // Session required → injects ctx.auth
        └── premiumProcedure  // Active Polar subscription required → injects
ctx.customer
```

The `protectedProcedure` middleware calls `auth.api.getSession({ headers })` on every request and throws `UNAUTHORIZED` if no session exists.

The `premiumProcedure` additionally calls `polarClient.customers.getStateExternal({ externalId: userId })` to verify an active subscription before allowing mutations like workflow creation and credential management.

Routers:

Router	Key Operations
`workflowsRouter`	`create`, `remove`, `update` (full canvas save), `updateName`, `getOne`, `getMany`, `execute`
`credentialsRouter`	`create`, `remove`, `update`, `getOne`, `getMany`, `getByType`
`executionsRouter`	`getOne`, `getMany` (with workflow join)

All routers are composed into the root `appRouter` in `src/trpc/routers/_app.ts`.

Server-side prefetching uses `createCallerFactory` to call tRPC procedures directly in Server Components, hydrating the TanStack Query cache before the page reaches the browser — eliminating loading states for initial data.

5.2 Authentication (better-auth)

Authentication is handled by **better-auth**, a newer full-stack auth library that works natively with

Next.js App Router.

Configured providers:

Provider	Method
Email / Password	Built-in, with auto sign-in on registration
GitHub OAuth	`GITHUB_CLIENT_ID` / `GITHUB_CLIENT_SECRET`
Google OAuth	`GOOGLE_CLIENT_ID` / `GOOGLE_CLIENT_SECRET`

Session flow:

1. `POST /api/auth/sign-in/email` or OAuth redirect
2. Session stored in the `session` table with a unique token
3. `auth.api.getSession({ headers })` reads the cookie on every server request
4. The `requireAuth` / `requireUnauth` helpers in `src/lib/auth-utils.ts` provide guards for layouts

Polar plugin is registered with better-auth, which automatically:

- Creates a Polar customer on user sign-up (`createCustomerOnSignUp: true`)
- Exposes `/api/auth/checkout` for initiating a Polar subscription checkout
- Exposes `/api/auth/portal` for redirecting to the Polar customer portal

The auth instance creates its **own** `PrismaClient` (separate from the global db singleton) to avoid conflicts with better-auth's internal model expectations.

5.3 Database Layer (Prisma + PostgreSQL)

Prisma is the ORM, generating type-safe client code from `prisma/schema.prisma`.

The generated client outputs to `src/generated/prisma/` so it is version-controlled and available at runtime without a separate generation step in dev.

Schema overview:

```
User ──< Session          (cascade delete sessions on user delete)
User ──< Account          (OAuth accounts; cascade delete)
User ──< Workflow         (cascade delete workflows on user delete)
User ──< Credential       (cascade delete credentials on user delete)
Workflow ──< Node         (cascade delete nodes on workflow delete)
Workflow ──< Connection   (cascade delete connections on workflow delete)
Workflow ──< Execution    (cascade delete executions on workflow delete)
Node ──< Connection       (fromNode/toNode; cascade delete connections on node
delete)
Node >── Credential       (optional FK — node references its credential)
```

Prisma singleton pattern prevents connection pool exhaustion in Next.js hot-reload:

```
// src/lib/db.ts
```

```
const globalForPrisma = global as unknown as { prisma: PrismaClient };
const prisma = globalForPrisma.prisma || new PrismaClient();
if (process.env.NODE_ENV !== "production") {
  globalForPrisma.prisma = prisma;
}
```

Key model details:

Model	Notable Fields
`Workflow`	`name`, `nodes[]`, `connections[]`, `executions[]`, `userId`
`Node`	`type: NodeType` (enum), `position: Json`, `data: Json` (node config), `credentialId?`
`Connection`	`fromNodeId`, `toNodeId`, `fromOutput`, `toInput` (both default to `"main"`)
`Execution`	`inngestEventId: String @unique`, `status: ExecutionStatus`, `output: Json?`, `error`, `errorStack`
`Credential`	`type: CredentialType`, `value?`, `email?`, `appPassword?`, SMTP fields, `instagramBusinessId?`

5.4 Credential Storage

Credentials are stored in plain text in the `Credential` table columns (no at-rest encryption at the application level). Different credential types use different columns:

Credential Type	Columns Used
OpenAI / Gemini / Anthropic / DeepSeek / Perplexity	`value` (API key)
Gmail	`email`, `appPassword` (Gmail App Password)
Custom SMTP	`smtpHost`, `smtpPort`, `smtpUser`, `smtpPassword`, `secure`
Instagram	`value` (access token), `instagramBusinessId`

Validation is enforced in the tRPC `credentialsRouter` via Zod `superRefine` — a separate `else if` branch per credential type ensures the correct fields are present before any DB write.

---

6. Workflow Execution Engine

This is the core of ConnectFlow and its most architecturally interesting system.

6.1 Inngest Functions & Durable Execution

Inngest is a durable execution platform. Instead of running workflow logic inside a plain HTTP handler that would time out or lose state on cold start, ConnectFlow sends an event to Inngest and Inngest drives the execution as a series of checkpointed **steps**.

The single Inngest function `execute-workflow` handles all workflow runs:

```
export const executeWorkflow = inngest.createFunction(
  {
    id: "execute-workflow",
    retries: process.env.NODE_ENV === "production" ? 3 : 0,
    onFailure: async ({ event }) => {
      // Mark execution as FAILED in DB
      await prisma.execution.update({
        where: { inngestEventId: event.data.event.id },
        data: { status: "FAILED", error: ..., errorStack: ... },
      });
    },
  },
  { event: "workflow/execute.workflow", channels: [...] },
  async ({ event, step, publish }) => { ... }
);
```

Why Inngest?

Challenge	Inngest Solution
Long-running workflows	Each `step.run()` is checkpointed; function can pause and resume across restarts
Retries	Built-in retry logic with `retries: 3` in production
Failure handling	`onFailure` callback updates the DB even when the function crashes
Real-time updates	Realtime channels stream status events to the browser mid-execution
AI integration	`step.ai.wrap()` instruments AI SDK calls with telemetry and checkpointing

The 5 execution steps:

```
1. create-execution     → INSERT INTO execution (workflowId, inngestEventId)
2. prepare-workflow     → SELECT workflow with nodes+connections,
                          topological sort, extract userId
                          (single DB query for both)
3. [per-node loop]      → For each node: getExecutor(type)(params)
                          Each executor may call step.run() internally
4. [implicit]           → Context accumulates across nodes
5. update-execution     → UPDATE execution SET status=SUCCESS, output=context
```

6.2 Topological Sort

Before execution, nodes are sorted into a deterministic execution order using **Kahn's algorithm** via the `toposort` library.

```
// src/inngest/utils.ts
export const topologicalSort = (nodes: Node[], connections: Connection[]):
Node[] => {
  if (connections.length === 0) return nodes;

  const edges: [string, string][] = connections.map(c => [c.fromNodeId,
c.toNodeId]);

  // Isolated nodes (no connections) are added as self-loops so toposort
includes them
  for (const node of nodes) {
    if (!connectedNodeIds.has(node.id)) {
      edges.push([node.id, node.id]);
    }
  }

  const sortedIds = [...new Set(toposort(edges))];
  // Cycle detection: toposort throws "Cyclic dependency" on circular graphs
  return sortedIds.map(id => nodeMap.get(id)!).filter(Boolean);
};
```

Cyclic dependencies are detected and surfaced as a `NonRetriableError` — the error propagates to Inngest's `onFailure` handler which marks the execution as `FAILED` without consuming any retry budget.

6.3 Executor Registry Pattern

Every node type is mapped to an executor function via the **executor registry**:

```
// src/features/executions/lib/executor-registry.ts
export const executorRegistry: Record<NodeType, NodeExecutor> = {
  [NodeType.ANTHROPIC]: anthropicExecutor,
  [NodeType.GEMINI]:    geminiExecutor,
  [NodeType.OPENAI]:    openAIExecutor,
  [NodeType.DISCORD]:   discordExecutor,
  [NodeType.GMAIL]:     gmailExecutor,
  // ...
};

export const getExecutor = (type: NodeType): NodeExecutor => {
  const executor = executorRegistry[type];
  if (!executor) throw new Error(`No executor found for node type: ${type}`);
  return executor;
};
```

NodeExecutor type contract:

```
type NodeExecutor<TData = Record<string, unknown>> = (
  params: {
    data: TData;          // Node configuration (from Node.data JSON)
    nodeId: string;       // Used for real-time status publishing
    userId: string;       // For scoped credential lookup
    context: WorkflowContext; // Accumulated output from prior nodes
    step: StepTools;      // Inngest step tools (step.run, step.ai.wrap, etc.)
    publish: Realtime.PublishFn; // For streaming status to browser
  }
) => Promise<WorkflowContext>; // Returns updated context
```

The `Record<NodeType, NodeExecutor>` type ensures at compile time that every enum value has a registered executor — no runtime surprises from missing entries.

6.4 Context Propagation

The **workflow context** is a plain `Record<string, any>` that flows through every node in order. Each executor receives the context, performs its work, and returns the context augmented with its output under its configured `variableName`:

```
Initial context: { stripe: { eventType: "payment_intent.succeeded", ... } }

After HTTP Request node (variableName="apiResult"):
{ stripe: {...}, apiResult: { httpResponse: { status: 200, data: {...} } } }

After Anthropic node (variableName="summary"):
{ stripe: {...}, apiResult: {...}, summary: { text: "Payment received..." } }

After Gmail node (variableName="emailResult"):
{ stripe: {...}, apiResult: {...}, summary: {...}, emailResult: { success: true,
messageId: "..." } }
```

The final `context` object is persisted to `Execution.output` as JSON at the end of the run.

Trigger-supplied data is injected at the start:

- Stripe webhook: `context.stripe = { eventId, eventType, timestamp, raw }`
- Google Form webhook: `context.googleFormData = { formId, responses, ... }`
- API trigger: `context.api = { ...userPayload }`

6.5 Handlebars Template Engine

Every text field in a node's configuration (prompts, email bodies, HTTP endpoints, etc.) supports **Handlebars template syntax**. This lets users reference prior node outputs dynamically:

```
Subject: Payment confirmed - {{ stripe.eventType }}
Body: Hi, your amount of {{ apiResult.httpResponse.data.amount }} has been
processed.
      AI Summary: {{ summary.text }}
```

A custom `json` helper is registered in every executor:

```
Handlebars.registerHelper("json", (context) =>
  new Handlebars.SafeString(JSON.stringify(context, null, 2))
);
```

This lets users embed structured data in prompts: `{{json apiResult}}`.

Templates are compiled and rendered at execution time, **after** all prior nodes have run, giving full access to accumulated context values.

---

7. Node Catalogue

7.1 Trigger Nodes

Trigger nodes are the entry point of every workflow. They receive external events and populate the

initial context.

Node	Trigger Mechanism	Initial Context Key
Manual Trigger	Button click in editor UI	(passes through existing context unchanged)
Stripe Trigger	Webhook at `/api/webhooks/stripe?workflowId=`	`context.stripe`
Google Form Trigger	Webhook at `/api/webhooks/google-form?workflowId=`	`context.googleFormData`
API Trigger	`POST /api/trigger` with `Authorization: Bearer cf_sk_...`	`context.api`

The webhook routes validate `workflowId` from the query string and call `sendWorkflowExecution` which fires the Inngest event `workflow/execute.workflow`.

The API Trigger uses AES-256-CBC encrypted keys (see §9).

7.2 Action Nodes

Action nodes consume and produce context values.

Node	Integration	Key Config Fields	Output Structure
HTTP Request	Any REST endpoint	`endpoint`, `method`, `body` (Handlebars), `variableName`	`{ httpResponse: { status, statusText, data } }`
Anthropic	Claude (via Vercel AI SDK)	`userPrompt`, `systemPrompt`, `credentialId`, `variableName`	`{ text }`
OpenAI	GPT-3.5-turbo (via Vercel AI SDK)	`userPrompt`, `systemPrompt`, `credentialId`, `variableName`	`{ text }`
Gemini	Gemini 2.0 Flash (via Vercel AI SDK)	`userPrompt`, `systemPrompt`, `credentialId`, `variableName`	`{ text }`
DeepSeek	deepseek-chat (via Vercel AI SDK)	`userPrompt`, `systemPrompt`, `credentialId`, `variableName`	`{ text }`
Perplexity	Sonar (via Vercel AI SDK)	`userPrompt`,	`{ text }`

Node	Integration	Key Config Fields	Output Structure
		`systemPrompt`, `credentialId`, `variableName`	
Gmail	Gmail SMTP via Nodemailer	`to`, `subject`, `body`, `credentialId`, `variableName`	`{ success, messageId, to }`
Custom SMTP	Any SMTP server via Nodemailer	`to`, `cc`, `bcc`, `subject`, `body`, `credentialId`, `variableName`	`{ success, messageId, to }`
Discord	Discord Incoming Webhook	`webhookUrl`, `content`, `username`, `variableName`	`{ messageContent }`
Instagram	Meta Graph API v16	`imageUrl`, `caption`, `credentialId`, `variableName`	`{ success, creationId, postId }`
Luma	Luma Dream Machine API	`imagePrompt`, `imageSize`, `apiCredentialId`, `variableName`	`{ imageUrl, prompt, generationId }`

AI node pattern — all five AI executors share the same structure:

1. Validate required fields
2. Compile Handlebars prompts against current context
3. Fetch credential from DB (scoped to `userId`)
4. Call `step.ai.wrap(name, generateText, { model, system, prompt })` — this checkpoints the AI call so retries don't re-invoke the model if the step already succeeded
5. Extract `steps[0].content[0].text` from the response
6. Return `{ ...context, [variableName]: { text } }`

---

8. Webhook Ingestion

External services send events to ConnectFlow's webhook endpoints. The workflow is identified via a `?workflowId=` query parameter on the URL.

```
POST /api/webhooks/stripe?workflowId=cm...
POST /api/webhooks/google-form?workflowId=cm...
```

Ingestion flow:

```
External Service ──POST──▶ /api/webhooks/<type>?workflowId=X
                                    |
                              Parse + validate body
                                    |
                           sendWorkflowExecution({
                             workflowId: X,
                             initialData: { stripe: {...} }
                           })
                                    |
                           inngest.send({
                             name: "workflow/execute.workflow",
                             data: { workflowId, initialData, ... },
                             id: createId()  // deduplicated by Inngest
                           })
                                    |
                        Inngest Cloud queues the job
                                    |
                     executeWorkflow function runs asynchronously
```

Stripe webhook body is mapped to:

```
{
  eventId: body.id,
  eventType: body.type,       // e.g. "payment_intent.succeeded"
  timestamp: body.created,
  livemode: body.livemode,
  raw: body.data?.object,     // Full Stripe event data object
}
```

---

9. API Trigger & Programmatic Execution

ConnectFlow exposes a programmatic HTTP trigger that allows any external system to start a workflow with a custom payload via a secret API key.

Key generation (server action):

```
// src/features/triggers/components/api-trigger/action.ts
const apiKey = generateAPIKey({
  userId: session.user.id,
  workflowId,
  triggerNodeId: nodeId,
});
// Returns: "cf_sk_<iv_hex>:<encrypted_hex>"
```

Encryption (`src/lib/api-key.ts`):

- Algorithm: **AES-256-CBC**
- Key derivation: `SHA-256(process.env.API_SECRET_KEY)` → fixed 32-byte key
- IV: cryptographically random 16 bytes, prepended to the ciphertext in the key string
- Payload: `JSON.stringify({ userId, workflowId, triggerNodeId? })`

Using the key:

```
POST /api/trigger
```

```
Authorization: Bearer cf_sk_<key>
Content-Type: application/json

{ "payload": { "orderId": "123", "amount": 9900 } }
```

Server-side verification:

```
// src/app/api/trigger/route.ts
const { userId, workflowId, triggerNodeId } = decryptAPIKey(apiKey);
// On any tamper or wrong key → throws → 401 response
await triggerAPIWorkflow({ userId, workflowId, triggerNodeId, payload });
```

The decrypted `userId` and `workflowId` are trusted server-side — no database lookup needed to authorize the request.

---

10. Monetisation & Subscriptions (Polar)

ConnectFlow gates key features behind an active subscription using **Polar**, an open-source merchant of record.

Integration points:

Location	Mechanism
`src/lib/auth.ts`	`polar({ client, createCustomerOnSignUp: true })` plugin — creates Polar customer on signup
`src/lib/polar.ts`	`new Polar({ accessToken, server: "sandbox" })` SDK client
`src/trpc/init.ts`	`premiumProcedure` — calls `polarClient.customers.getStateExternal` on every request
`src/components/app-sidebar.tsx`	Polls subscription every 5 s if inactive; shows "Upgrade to Pro" button
`src/components/upgrade-modal.tsx`	Modal triggered when a non-subscriber tries a premium action

Subscription-gated operations:

- `workflows.create` → requires active subscription (`premiumProcedure`)
- `credentials.create` → requires active subscription (`premiumProcedure`)

Checkout flow:

```
User clicks "Upgrade to Pro"
  → authClient.checkout({ slug: "pro" })
  → Redirects to Polar hosted checkout
  → On success → Polar calls /api/auth/webhook (better-auth Polar plugin
handles)
  → Subscription record created
  → User redirected to POLAR_SUCCESS_URL
  → App polls subscription every 5 s → detects active → updates UI
```

---

11. Error Handling & Observability

Application Errors

- **Inngest `NonRetriableError`** — thrown from executors for validation failures (missing fields, invalid credentials). Inngest immediately stops retrying and calls `onFailure`, which marks the execution `FAILED` with the error message.
- **Retriable errors** — any other thrown error is retried up to 3 times in production with Inngest's exponential back-off. After exhaustion, `onFailure` fires.
- **Zod validation** — all tRPC inputs are validated via Zod schemas before reaching resolvers. Invalid input returns a `BAD_REQUEST` tRPC error automatically.
- **TRPCError** — `UNAUTHORIZED` (no session), `FORBIDDEN` (no subscription), `NOT_FOUND` (resource missing or wrong owner).

Sentry

`@sentry/nextjs` is configured via `src/instrumentation.ts` (Node.js) and `src/instrumentation-client.ts` (browser). Sentry captures:

- Unhandled exceptions in Server Components and Route Handlers
- Client-side JavaScript errors
- Performance traces

The `/sentry-example-page` and `/api/sentry-example-api` routes exist for validating the Sentry setup.

Execution Audit Trail

Every workflow run persists a complete record:

```
Execution {
  id, workflowId, inngestEventId,
  status: RUNNING → SUCCESS | FAILED,
  startedAt, completedAt,
  output: Json    // Final accumulated context (all node outputs)
  error: Text,    // Set only on FAILED
  errorStack: Text
}
```

Users can inspect any past execution through the Executions dashboard, drilling into the full output JSON.

---

12. Security Model

Concern	Approach
Authentication	Session cookie managed by better-auth; OAuth via GitHub/Google
Authorization	Every tRPC mutation filters by `userId` in WHERE clauses — users can only access their own resources
Premium gating	`premiumProcedure` verifies subscription on every protected mutation at the server
API Key security	Keys are AES-256-CBC encrypted; decryption provides `userId` + `workflowId` without any DB lookup; tampered keys throw and return 401
Credential isolation	Executor credential lookups always include `userId` in the WHERE clause: `prisma.credential.findUnique({ where: { id, userId } })` — prevents credential theft between users
Webhook auth	Webhooks identify their workflow via `workflowId` in the query string only — no Stripe signature verification in the current implementation
Secret management	All secrets via environment variables; `API_SECRET_KEY` throws on startup if unset; no hardcoded fallback values
Sensitive log removal	No API keys or secrets are ever logged to stdout

---

13. Data Flow Walkthrough — End to End

Scenario: A Stripe `payment_intent.succeeded` event triggers a workflow that calls the OpenAI API to generate a payment confirmation message, then emails it via Gmail.

```
1. STRIPE WEBHOOK
   POST /api/webhooks/stripe?workflowId=cm_abc123
   Body: { id, type: "payment_intent.succeeded", data: { object: {...} } }
   → Extracts: { eventType, raw: { amount, currency, ... } }
   → inngest.send("workflow/execute.workflow", { workflowId, initialData:
{ stripe: {...} } })
   → Returns HTTP 200 immediately

2. INNGEST QUEUES JOB
   Event received by Inngest Cloud
   → executeWorkflow function invoked

3. STEP: create-execution
   INSERT INTO Execution (workflowId, inngestEventId)
   status = RUNNING

4. STEP: prepare-workflow
```

```
   SELECT Workflow with nodes + connections
   → topologicalSort: [StripeTrigger, OpenAI, Gmail]
   → userId extracted from workflow row

5. NODE 1: StripeTriggerExecutor
   publish("stripe-execution", status: "loading")
   step.run → returns context unchanged
   publish("stripe-execution", status: "success")
   context = { stripe: { eventType: "payment_intent.succeeded", raw: { amount:
9900 } } }

   [Browser receives SSE: Stripe node turns green]

6. NODE 2: OpenAIExecutor
   publish("openai-execution", status: "loading")
   step.run("get-credential") → SELECT credential WHERE id=X AND userId=Y
   Compile Handlebars: prompt = "Write a confirmation for payment of $
{{stripe.raw.amount}}"
   step.ai.wrap("openai-generate-text", generateText, { model: "gpt-3.5-
turbo", ... })
   → AI response: "Your payment of $99.00 has been confirmed..."
   publish("openai-execution", status: "success")
   context = { stripe: {...}, aiMessage: { text: "Your payment of $99.00..." } }

   [Browser receives SSE: OpenAI node turns green]

7. NODE 3: GmailExecutor
   publish("gmail-execution", status: "loading")
   step.run("get-credential") → SELECT credential WHERE id=X AND userId=Y
   Compile: to = "customer@example.com", subject = "Payment Confirmed",
            body = "{{aiMessage.text}}"
   step.run("gmail-send") → nodemailer.sendMail(...)
   publish("gmail-execution", status: "success")
   context = { ..., emailResult: { success: true, messageId: "..." } }

   [Browser receives SSE: Gmail node turns green]

8. STEP: update-execution
   UPDATE Execution
   SET status = SUCCESS,
       completedAt = NOW(),
       output = { stripe: {...}, aiMessage: {...}, emailResult: {...} }
```

---

14. Environment Variables Reference

Variable	Required	Description
DATABASE_URL		PostgreSQL connection string
BETTER_AUTH_SECRET		Secret for signing better-auth sessions
BETTER_AUTH_URL		Base URL of the app (e.g. https://connectflow.app)
GITHUB_CLIENT_ID		GitHub OAuth app client ID
GITHUB_CLIENT_SECRET		GitHub OAuth app client secret
GOOGLE_CLIENT_ID		Google OAuth app client ID

Variable	Required	Description
`GOOGLE_CLIENT_SECRET`		Google OAuth app client secret
`INNGEST_EVENT_KEY`		Inngest event signing key
`INNGEST_SIGNING_KEY`		Inngest webhook signing key
`API_SECRET_KEY`		Random string for AES-256 API key encryption (fails on startup if missing)
`POLAR_ACCESS_TOKEN`		Polar SDK access token (use sandbox token in dev)
`POLAR_SUCCESS_URL`		Redirect URL after successful Polar checkout
`NEXT_PUBLIC_APP_URL`		Public base URL (used by Inngest serve endpoint)
`SENTRY_DSN`	⚙	Sentry project DSN for error reporting
`NGROK_URL`		ngrok static domain for local webhook testing

---

15. Scripts & Tooling

Script	Command	Description
Development	`npm run dev`	Next.js dev server with Turbopack
Inngest Dev	`npm run inngest:dev`	Local Inngest dev server (connects to local app)
All Together	`npm run dev:all`	`mprocs` runs Next.js + Inngest + ngrok in parallel
ngrok	`npm run ngrok:dev`	Expose local server for webhook testing
Build	`npm run build`	Next.js production build (Turbopack)
Start	`npm run start`	Production server
Vercel Build	`vercel-build`	`prisma generate` → `prisma migrate deploy` → `next build`
Lint	`npm run lint`	Biome check (lint + format check)
Format	`npm run format`	Biome format (auto-fix)
Prisma	`prisma studio`	GUI database browser
Prisma	`prisma migrate dev`	Create and apply a new

Script	Command	Description
		migration

Biome replaces both ESLint and Prettier with a single fast Rust-based tool, configured in `biome.json`.

mprocs orchestrates multiple terminal processes in a single pane, letting you run the Next.js dev server, the Inngest dev UI, and ngrok simultaneously without juggling tabs.

Generated from the live ConnectFlow codebase — May 2026