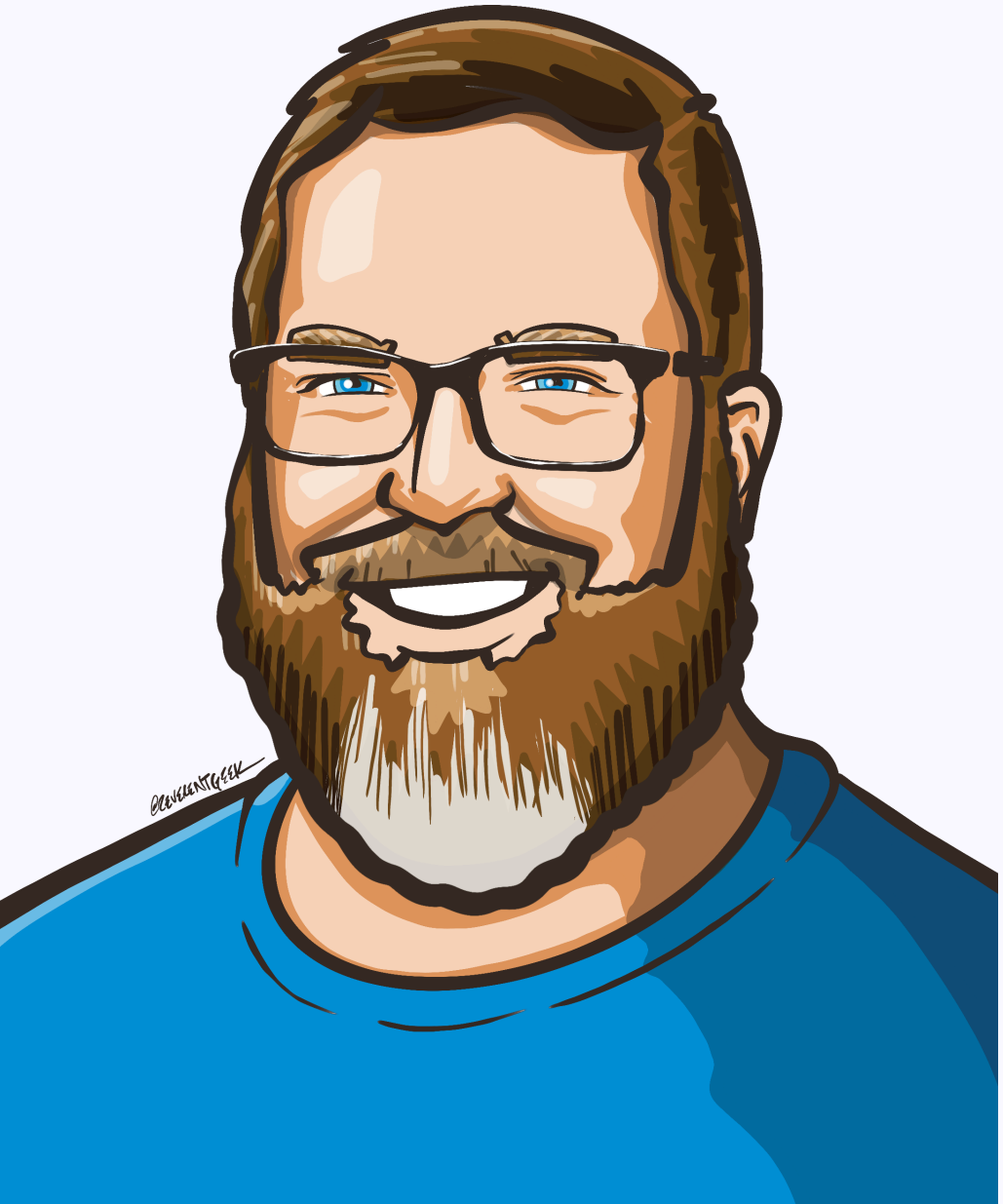
Polyglot Aspire

One app model.
Any AppHost.
Any workload.

Chris Ayers

https://github.com/codebytes/aspire-polyglot



Chris Ayers

Principal Software Engineer
Azure EngOps AzRel
Microsoft

BlueSky: @chris-ayers.com
LinkedIn: - chris-l-ayers
Blog: https://chris-ayers.com/
GitHub: Codebytes
Mastodon: @Chrisayers@hachyderm.io

The Polyglot Problem

> *Modern apps are an **orchestra with no conductor**, and for polyglot teams, every section is reading from a different score.*

Your team doesn't use one language. It uses **five**.

Your stack today

- Python ML services
- Go microservices
- Java Spring Boot APIs
- TypeScript/React frontends
- .NET backend APIs

The question: How do you orchestrate, observe, and wire all of this **from one place**?

The Orchestration Pain

Five stacks. Five toolchains. Zero shared model.

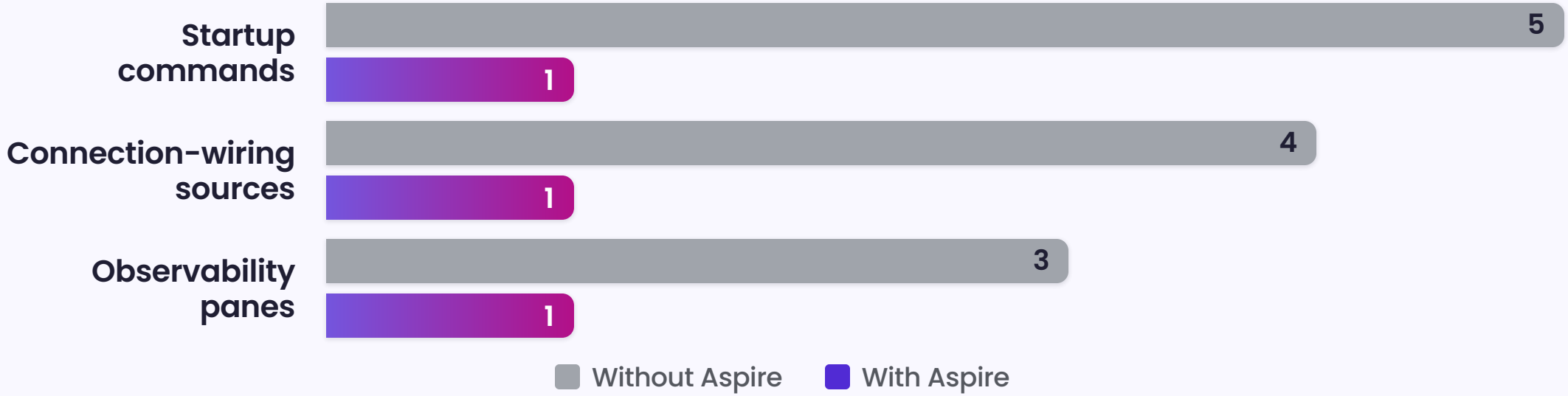
- 🚢 **Docker Compose**: explicit endpoint, dependency, and health-check configuration; telemetry is separate
- 🗒️ **Config sprawl**: `.env`, YAML, `application.properties`, `appsettings.json`, each with its own format
- 🔍 **No unified observability**: good luck tracing a request across four services in three runtimes
- 📜 **15-step READMEs**: "just `docker-compose up`" never works first time

Aspire: The Polyglot Answer

Aspire is an agent-ready, code-first tool to compose, debug, and deploy any distributed app.

What Collapses Into One

An illustrative five-service team—not a measured benchmark:



The Four Pillars

POLYGLOT ASPIRE

Aspire CLI

CONTROL PLANE

One command set to start, inspect, and deploy the stack.

Aspire AppHost

STACK IN CODE

Services, dependencies, and endpoints declared together in code.

Aspire Dashboard

APP AT A GLANCE

Resource state, logs, traces, metrics, and interactive terminals.

Aspire Integrations

BUILDING BLOCKS

100+ packages for data, messaging, AI, and clouds.

One Orchestrator. Every Language.

Workload runtimes

Python · Go · Java

TypeScript · .NET

Aspire AppHost

C# · TypeScript

Python · Go · Java (preview)

one run · one model

Same capabilities

Orchestration

Service discovery

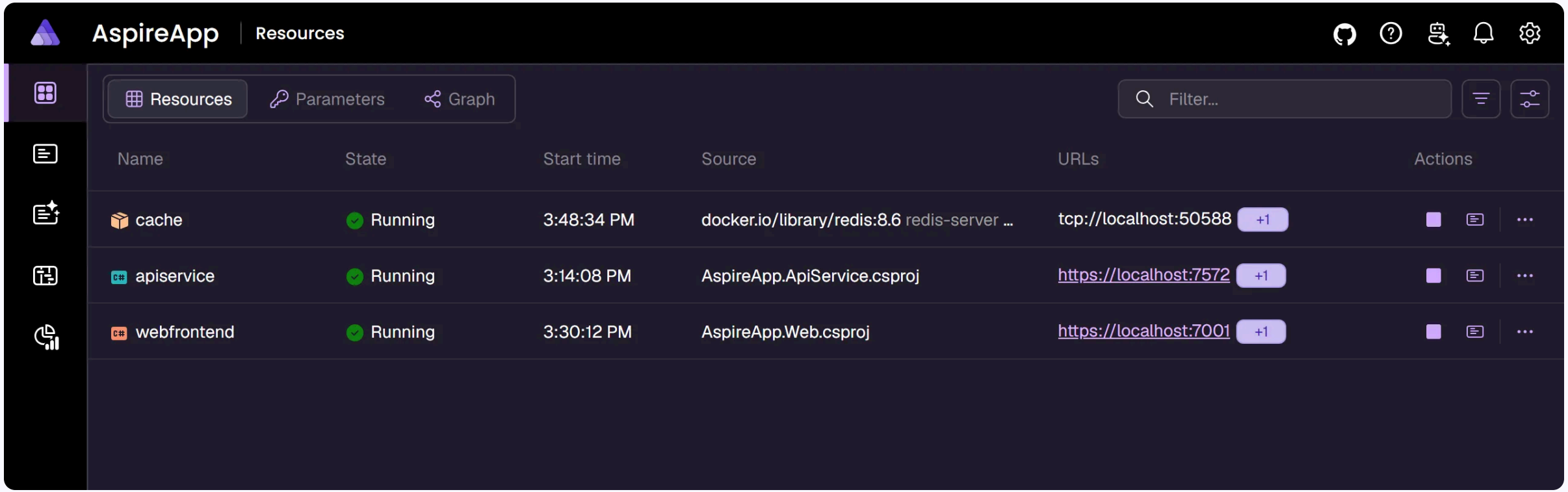
OpenTelemetry

AppHost Language ≠ Workload Language

One C# AppHost can wire Python, React, and .NET:

```
var builder = DistributedApplication.CreateBuilder(args);

var cache = builder.AddRedis("cache");
var ml = builder.AddUvicornApp("ml", "../python", "main:app")
    .WithHttpEndpoint(env: "PORT")
    .WithReference(cache);

builder.AddViteApp("web", "../react").WithReference(ml);
builder.AddProject<Projects.Api>("api").WithReference(cache);
builder.Build().Run();
```

The Dashboard: One View for Everything

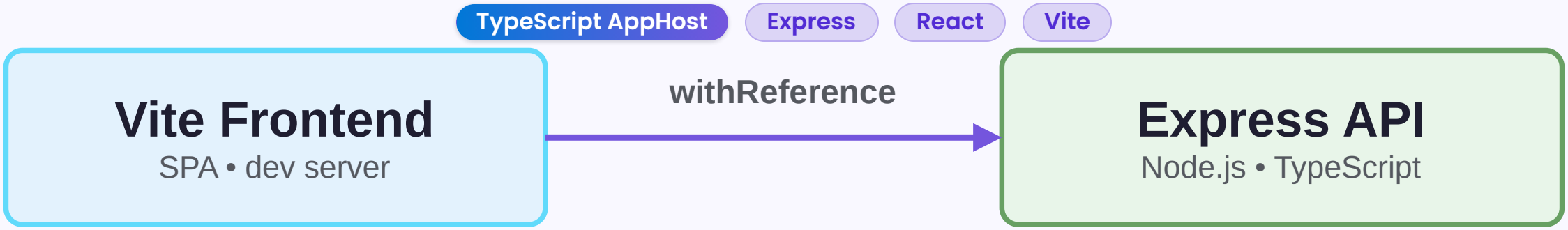


Source: Microsoft Aspire 13.5 release notes

13.5: Refreshed resource views, clearer health states, faster filtering, and sharper telemetry search.

Add OpenTelemetry to your services; Aspire supplies `OTEL_EXPORTER_OTLP_ENDPOINT`.

Demo: TypeScript Starter

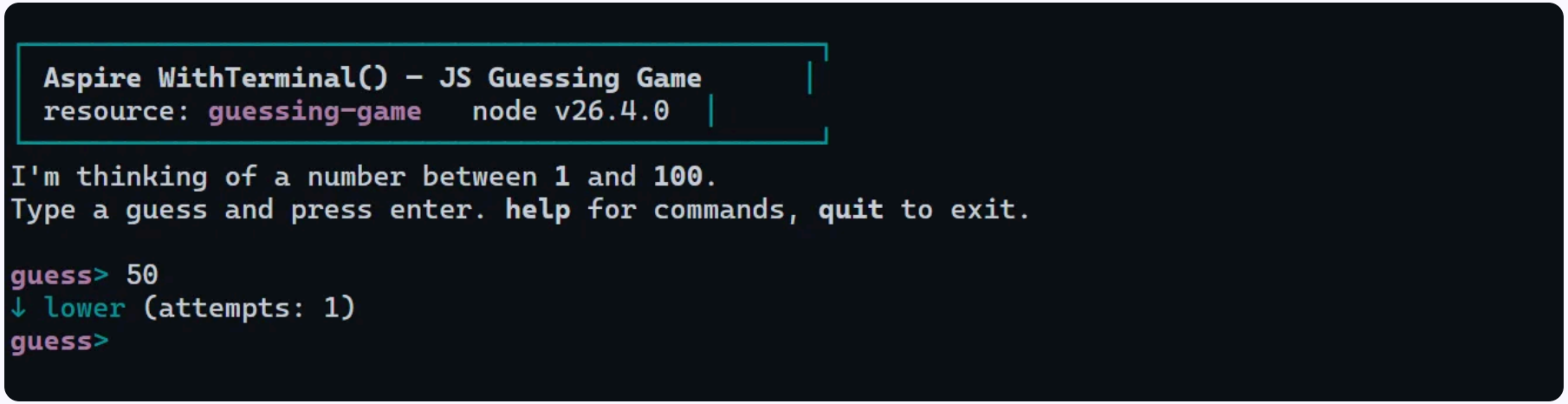


Aspire injects the Express URL into Vite and bundles for publish

Live: Refresh the forecast → switch °F / °C → inspect an API trace.

Not Everything Is an HTTP Service

Experimental in 13.5: REPLs, shells, and TUIs are resources too.



Cropped from the Microsoft Aspire 13.5 announcement

Enable `.WithTerminal()` / `.withTerminal()` on a resource.

Real input and output. One session, multiple viewers.

```
cd samples/terminal-demo
aspire terminal ps
aspire terminal attach node-repl
```

Observability Without Rewrites

Already emitting OpenTelemetry? Add a standalone dashboard.

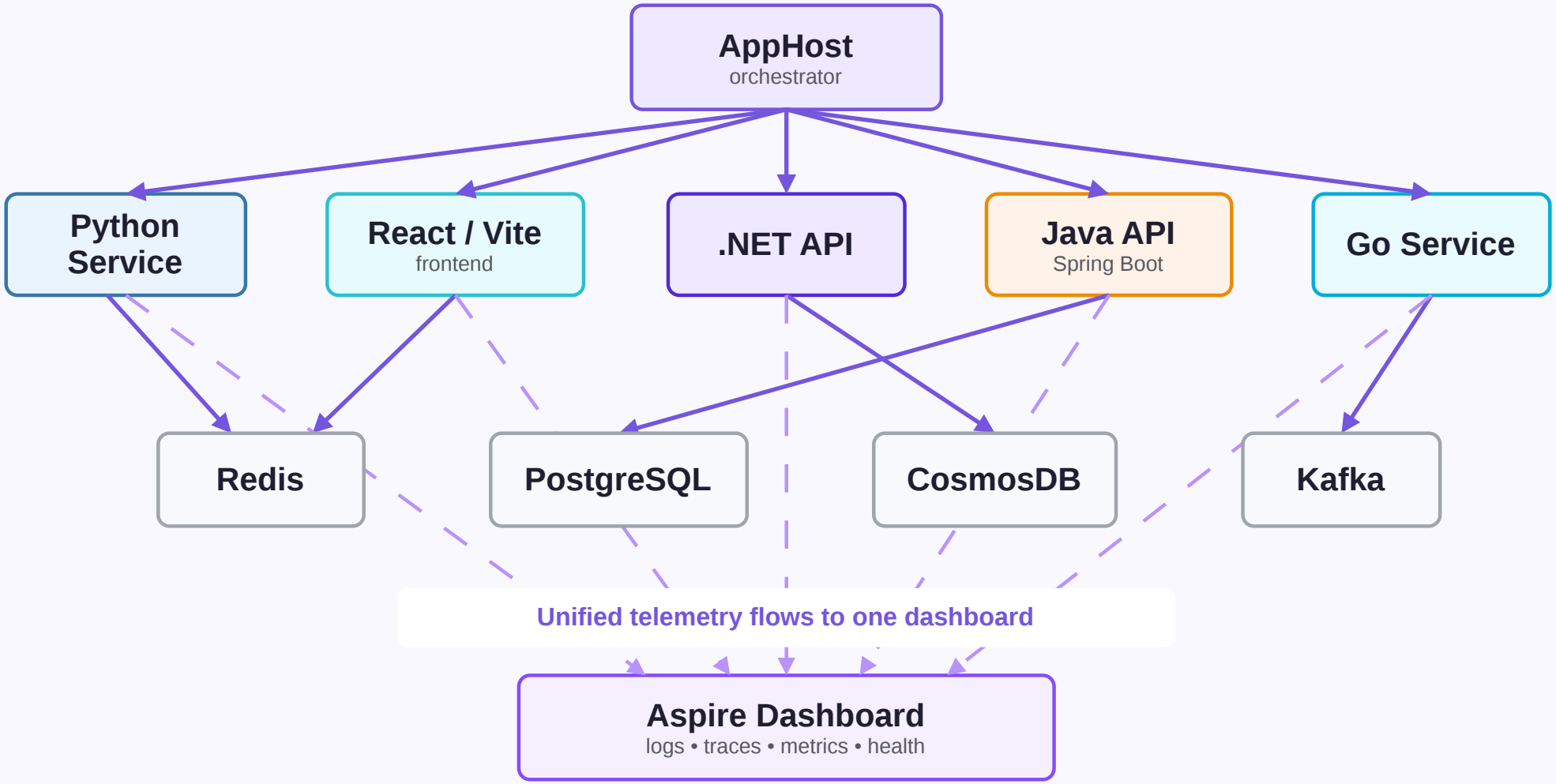
```
docker run --rm -d --name aspire-demo-dashboard \
  -p 127.0.0.1:18888:18888 \
  -p 127.0.0.1:4317:18889 -p 127.0.0.1:4318:18890 \
  mcr.microsoft.com/dotnet/aspire-dashboard:latest

docker logs aspire-demo-dashboard
```

Open the login URL from the logs. UI: `18888` · OTLP/gRPC: `4317` · OTLP/HTTP: `4318`

No .NET app required. Adopt the AppHost when you also need service wiring and lifecycle.

Architecture Overview

AppHost orchestrates everything: DCP manages processes, Dashboard collects telemetry



How It Works

The patterns that make polyglot orchestration possible

Service Discovery: The Pattern

Aspire injects service endpoints as environment variables:

```
# Pattern: services__<name>__<protocol>__<index>
services__api__http__0=http://localhost:5000
services__frontend__http__0=http://localhost:3000
```

Why double underscore? `__` represents configuration hierarchy using shell-friendly names.

No hardcoded service addresses. Application configuration and toolchains stay in place.

Connection Strings: The Pattern

Connection strings use `ConnectionStrings__<resource>`:

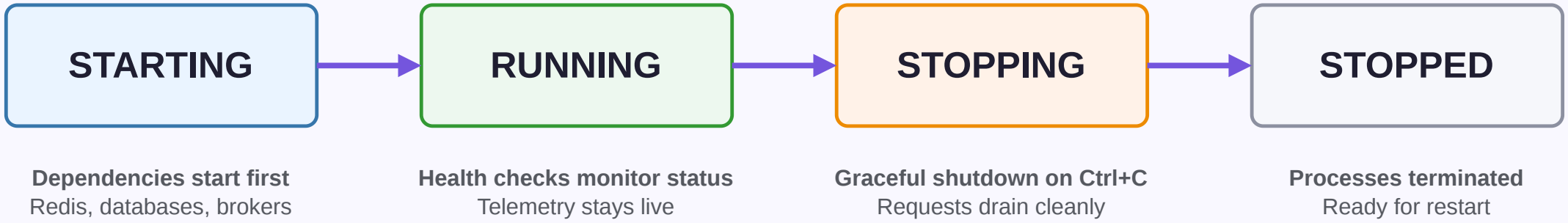
```
ConnectionStrings__messaging=localhost:9092
ConnectionStrings__recipesdb=<provider connection string>
```

Typed Redis resources also expose `CACHE_URI`. Use the complete URI:

```
import os
import redis

client = redis.Redis.from_url(os.environ["CACHE_URI"])
```

Case matters in Python/Linux. A connection string is not always a URI.

Resource Lifecycle Management



Aspire keeps startup order, monitoring, and shutdown coordinated across the whole app.

Startup Gating: `WaitFor(...)` holds dependents until a resource is ready
Health Monitoring: `WithHttpHealthCheck("/health")` reports readiness and status
Graceful Shutdown: Clean termination of all processes
Backed by 100+ integrations: Postgres, Redis, Kafka, Cosmos, OpenAI, Ollama, and your own containers all participate in the same lifecycle.

The AppHost: C#

Write your AppHost in the language your team knows. Here's C#:

```
var builder = DistributedApplication
    .CreateBuilder(args);

var redis = builder.AddRedis("cache");

builder.AddPythonApp("api", "../api", "app.py")
       .WithReference(redis)
       .WithHttpEndpoint(env: "PORT");

builder.Build().Run();
```

100+ integrations like Redis, Azure, Kafka, MongoDB, and PostgreSQL are available out of the box.

The AppHost: TypeScript

GA since 13.4. Aspire 13.5 closes more parity gaps with C#.

```
import { createBuilder } from "./.aspire/modules/aspire.mjs";

const builder = await createBuilder();

const cache = await builder.addRedis("cache");
const api = await builder.addPythonApp("api", "../api", "app.py")
  .withReference(cache);

await builder.addViteApp("web", "../web").withReference(api);
await builder.build().run();
```

Same integration model via ATS. 13.5 adds custom health checks, container files, HTTPS certificates, and richer interactions.

Start with a Workload Integration

Keep the workload's toolchain. Let Aspire model how it runs.

- 🐍 Python / ASGI → `AddPythonApp()` · `AddUvicornApp()`
- 🟦 JavaScript / TypeScript → `AddJavaScriptApp()` · `AddNodeApp()`
- ⚡ Vite frontend → `AddViteApp()`
- 💜 .NET project → `AddProject<T>()`
- 🐹 Go / Bun → `AddGoApp()` · `AddBunApp()`

Prefer a typed integration when one fits.

Bring the Rest of Your Stack

More integrations

- Deno: `AddDenoApp()` / `AddDenoTask()`
- Spring Boot: `AddSpringApp()`
- Blazor WASM: `AddBlazorWasmProject()`

Deno and Spring: **Community Toolkit**.
Blazor hosting: **preview**.

Universal options

- `AddDockerfile()` — build an image
- `AddContainer()` — use an image
- `AddExecutable()` — run a process

If it runs, it can join the model.

Infrastructure joins the same graph: **Redis · PostgreSQL · Kafka · Cosmos DB**.

Wire Connections and Readiness

```
.WithReference(redis)            // pass connection info
.WaitFor(postgres)               // wait for readiness
.WithHttpEndpoint(env: "PORT")    // expose HTTP
.WithExternalHttpEndpoints()     // public ingress
.WithHttpHealthCheck("/health") // readiness probe
```

A reference supplies configuration. A wait supplies startup ordering.

Practical Setup

Repeatable workflows. Familiar tools. Agents with live context.

Turn README Steps into Resource Commands

Define a workflow once. Use it from the dashboard or CLI.

Quotes Board: one custom `seed` command, two interfaces.

```
cd samples/dotnet-react-postgres
aspire resource api seed --help
aspire resource api seed --count 5
```

Dashboard collects named inputs

CLI accepts named options

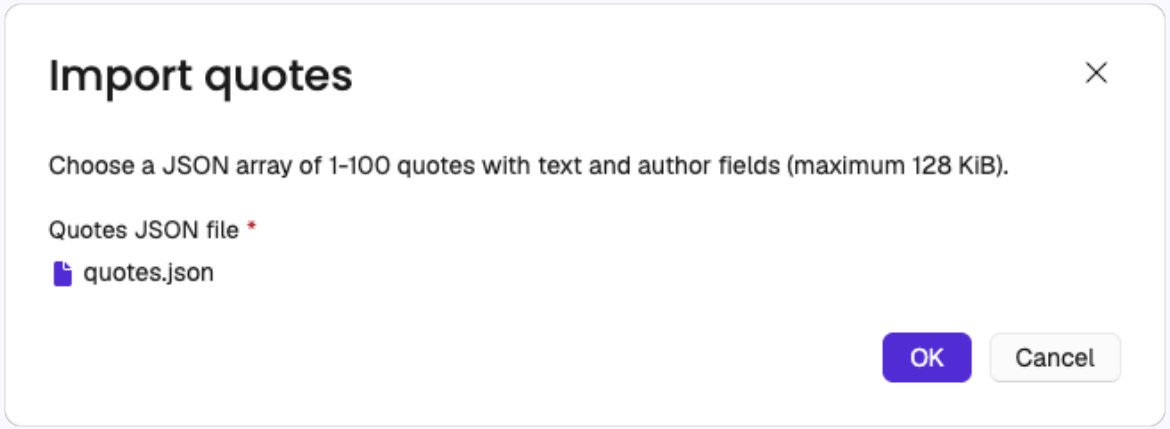
Stable in 13.5: command arguments and core prompts.

Let a Command Ask for a File

A guided import

- Choose `quotes.json`
- Validate content and size
- Import the whole batch

Stable: file inputs.
Experimental: progress dialogs.



Live Quotes Board sample · Aspire dashboard

Demo: Quotes Board Commands

C# AppHost | React | PostgreSQL

Seed from the terminal. Import from the dashboard.

```
aspire resource api seed --count 5
```

Seed quotes one named count argument

Import quotes choose demo/quotes.json

Click **Refresh quotes** → see new entries and the updated count.

Same application. Two repeatable workflows.

Create Your First App

Install with a familiar package manager: get.aspire.dev

```
npm install -g @microsoft/aspire-cli
aspire new aspire-ts-starter -n my-app
cd my-app
aspire run
```

One coherent path: install → scaffold → run.

Work with a Running Stack

Inspect the model and its resources; do not guess ports or processes.

```
aspire start      # Background
aspire ps         # AppHost summaries
aspire describe   # Resource details
aspire doctor     # Environment
aspire logs       # Console output
aspire stop       # Stop the app
```

`ps` summarizes **AppHosts**. `describe` shows **resources**.

Agent-Ready CLI

Skills for workflow. MCP for runtime context.
Start with Aspire skills. Add MCP when the agent needs live application data.

🧭 Aspire skills

Teach agents the CLI, AppHost conventions, and debugging workflows.

Install project guidance with `aspire agent init`.

🔌 Aspire MCP

The agent starts `aspire agent mcp` as a local **stdio** subprocess.

Read resource state, logs, and traces; execute resource commands.

One runtime view across Python, Go, Java, Node.js, and .NET.

Wire It Up in 30 Seconds

Install skills first; select the MCP server when you need runtime access.

In your AppHost directory

```
aspire agent init
# Select skills; optionally MCP
aspire run
```

Open your configured coding agent; it launches MCP on demand.

Generated VS Code config

```
{ "servers": {
    "aspire": {
      "type": "stdio",
      "command": "aspire",
      "args": ["agent", "mcp"]
    }
} }
```

More Polyglot Demos

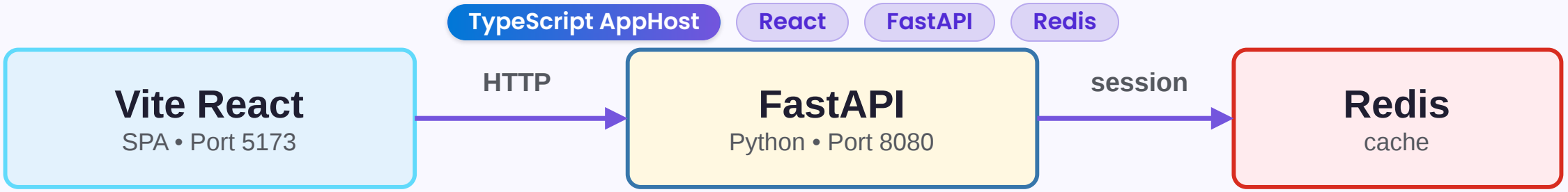
From the starter to mixed-language workflows: **16 verified samples**

Five Languages, One Dashboard

Across 16 samples: an authored AppHost or workload in each language

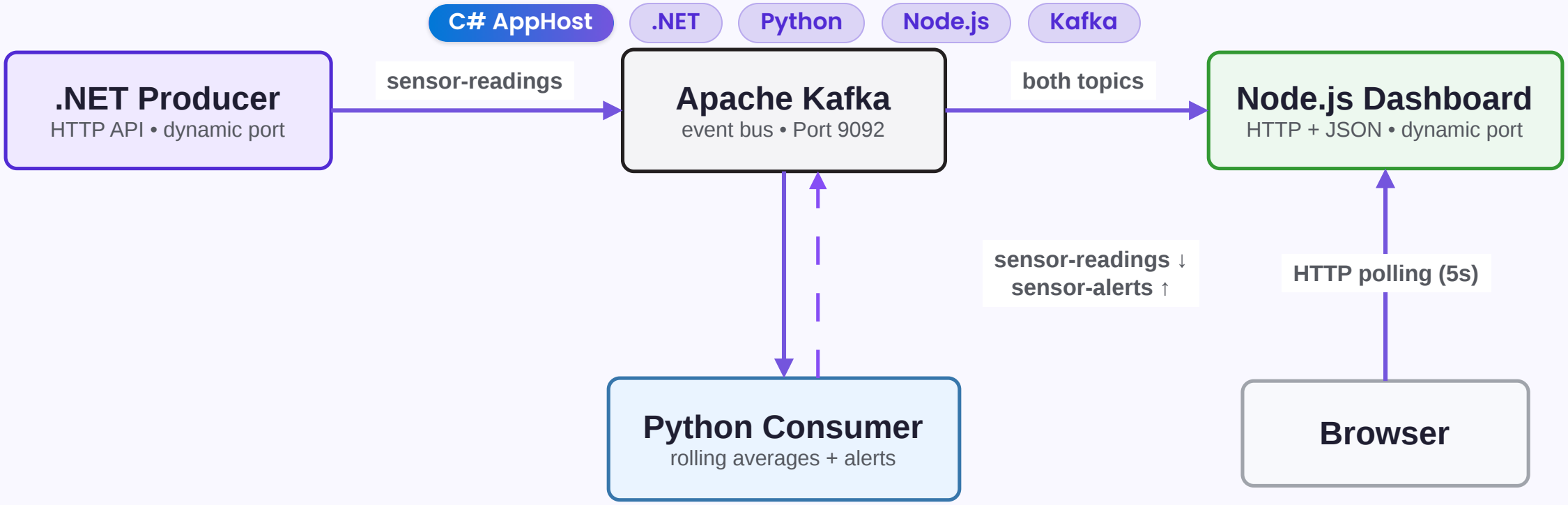
5 languages · **16** samples · **1** dashboard experience

Demo: Vite React + FastAPI



Aspire orchestrates Redis, FastAPI, and React via Dockerfiles

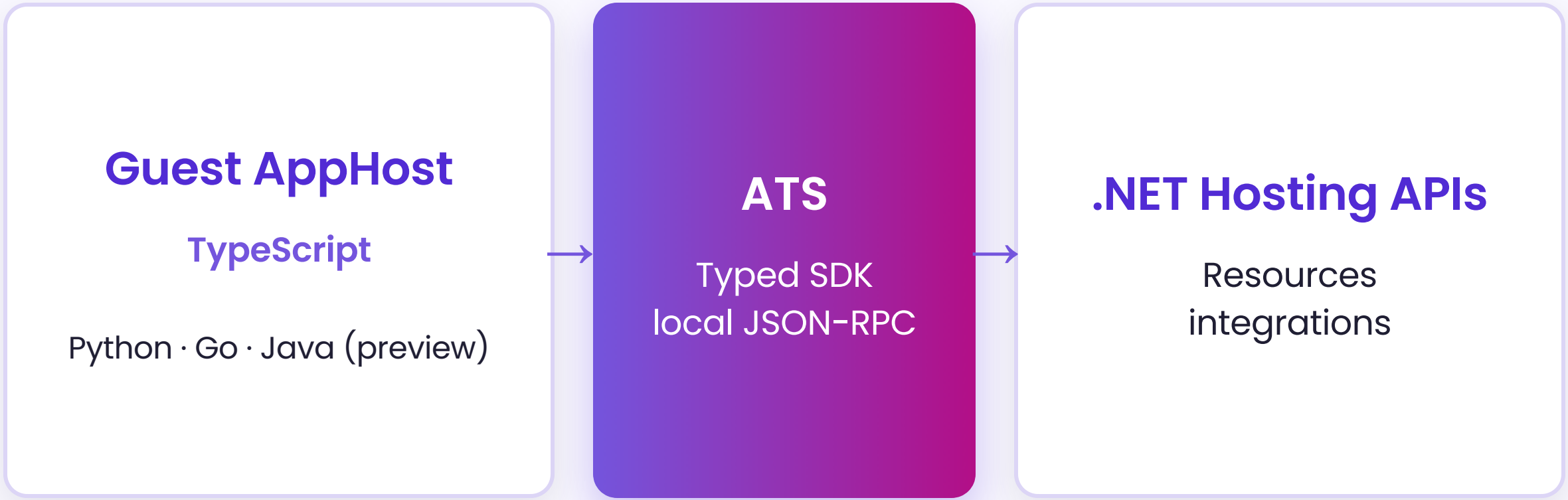
Demo: Polyglot Event Stream



Kafka carries readings and alerts; the browser polls every 5s

ATS: One Model, Multiple Languages

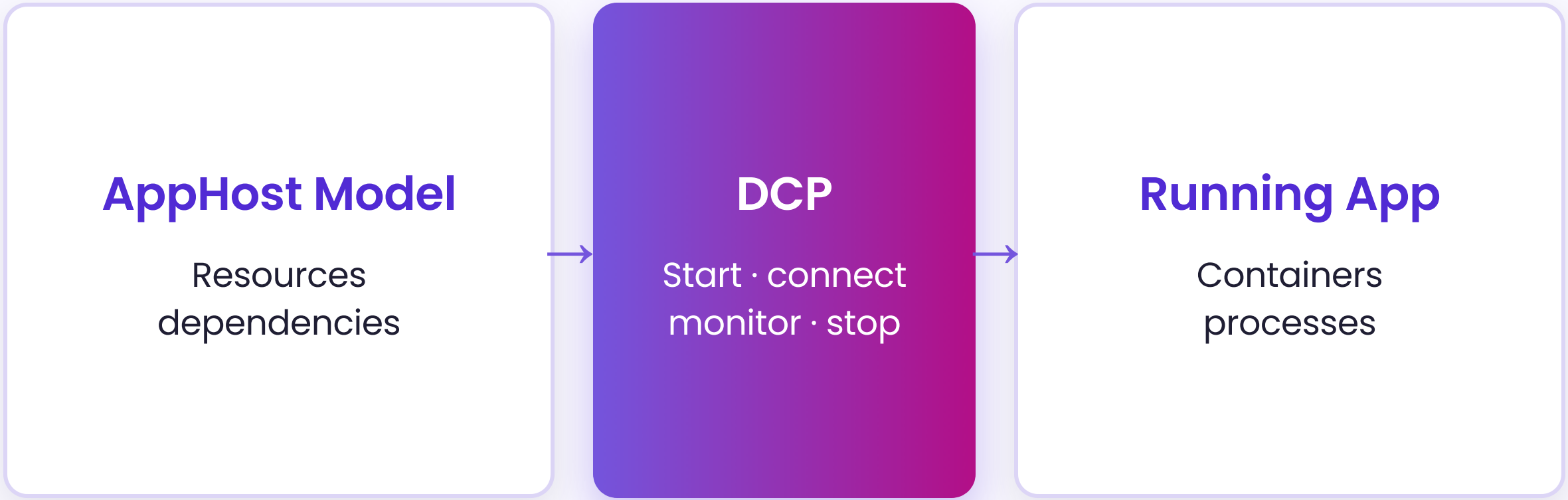
The **Aspire Type System** exposes hosting APIs to guest AppHost languages.



Generated SDKs live in `.aspire/modules/`.

DCP: Turn the Model into Processes

The **Developer Control Plane** runs the local resource graph.



Development-time only. Production uses the selected deployment target.

Same Model. Explicit Deployment Target.

Add a target integration and resource to the AppHost:

```
#:package Aspire.Hosting.Docker@13.5.3
// Alongside the API, cache, and their references:
builder.AddDockerComposeEnvironment("compose");
```

`aspire run`	local development
`aspire deploy`	configured target

Reuse the app model; select where it runs.

Production Choices Stay Explicit

Where will it run?

- Deployment platform
- Image registry and credentials

What does it need?

- Persistent storage
- Ingress, certificates, and access

13.5 preview: Kubernetes persistent-volume APIs (`ASPIRECOMPUTE002`).

`aspire publish` emits artifacts; `aspire do` runs named pipeline steps.

A Verified Publish Path: Go + Redis

The same model was tested locally and as a published Compose stack.

```
# From samples/go-redis-compose
aspire publish -o compose-output
docker build -t hitcounter-api:local src/api
cd compose-output
API_IMAGE=hitcounter-api:local docker compose up -d
docker compose port api 8080
```

Open `/api/hits` on the reported host port. Refresh → the Redis counter increases.

Verified: generated artifacts, working API, and counter retention across API restart.

Demo: AppHost to Azure Bicep

C# AppHost | Angular + .NET | Container Apps | Cosmos DB

```
cd samples/dotnet-angular-cosmos
aspire publish --apphost AppHost/AppHost.csproj \
  -o aspire-output/bicep --non-interactive
```

Inspect infrastructure + app modules

Compile 8 Bicep templates → ARM JSON

Publish is not deploy. No Azure resources are created.

Wrap-Up

Key Takeaways

🎯 **Two independent choices, one model**: Pick the AppHost language and workload runtimes separately

📊 **Unified observability out of the box**: One dashboard for logs, traces, and metrics across all services via OpenTelemetry

🚀 **One path to production**: the same model drives local development and deployment

Resources

Links

- 🌐 Aspire docs
- 🐙 Aspire source
- 🧪 Polyglot demos
- 🆕 Aspire 13.5
- 🧰 Aspire Community Toolkit
- 💬 Aspire Discord

Follow Chris Ayers

chris-ayers.com

Questions?