

# Manifest — Cursor Build Prompt (v2, research-verified)

## What is Manifest

A web/mobile app that wraps Charmbracelet's Crush (<https://github.com/charmbracelet/crush>) in a consumer chat UI. Users paste photos, share links, describe what they want — and get a live shareable URL. No terminal, no code, no deploy buttons. Under the hood: Crush runs on a per-user VM with pre-wired MCP tools and Docker-based deploy. DeepSeek V4 Flash is the default LLM.

---

## Crush Integration — Verified Behavior

Everything below is sourced from Crush's actual codebase, DeepWiki analysis, and GitHub issues. Do NOT guess at Crush behavior — use these facts.

### Source tree (confirmed from internal/ directory listing)

```
internal/
├─ agent/      # SessionAgent, Coordinator, tools, prompts, templates
├─ app/        # App orchestrator: DB, config, agents, LSP, MCP, events
├─ backend/    # Backend server implementation
├─ client/     # Client for connecting to backend server
├─ server/     # Server mode implementation
├─ proto/      # Protocol definitions (likely protobuf or JSON protocol)
├─ cmd/        # CLI commands: root, run, login, models, stats, sessions
├─ config/     # Config struct, crush.json loading, provider resolution
├─ db/         # SQLite persistence, sqlc queries, migrations
├─ mcp/        # MCP client integration
├─ hooks/      # User-defined shell hooks on tool events
├─ lsp/        # LSP manager and client
├─ permission/ # Permission system (--yolo bypasses)
├─ pubsub/     # Internal event bus
├─ session/    # Session service
├─ ...
```

The existence of `internal/backend/`, `internal/client/`, `internal/server/`, and `internal/proto/` confirms Crush has a client-server architecture already built.

## CRUSH\_CLIENT\_SERVER mode

When the environment variable `CRUSH_CLIENT_SERVER` is enabled:

- `crush run` uses `runNonInteractive` to **stream events from a remote server** (internal/cmd/run.go:155-163)
- Without it, `crush run` calls `appWs.App().RunNonInteractive` locally (internal/cmd/run.go:139)

This means Crush already has the server-side agent loop we need. The `internal/server/` package runs Crush as a backend daemon, and `internal/client/` connects to it. The `internal/proto/` package defines the wire protocol between them.

**This is the integration point for Manifest.** Instead of wrapping Crush as a subprocess with stdin/stdout, we likely connect to Crush's built-in server mode.

### Approach: Two strategies, pick after inspection

#### Strategy A (preferred): Use Crush's native server mode

1. Run `crush` in server mode on the VM (investigate `internal/server/`) — likely started via env var or flag)
2. The agent-relay connects as a Crush client using the protocol in `internal/proto/`
3. Multi-turn, streaming, tool calls all come through the existing protocol
4. This is what Charm built for their own IDE/extension use cases

#### Strategy B (fallback): Repeated `crush run` with session persistence

1. Each user message = one `crush run --yolo "message"` invocation
2. Sessions persist to SQLite in `.crush/crush.db`
3. Multi-turn works IF `crush run` supports `--session <id>` to resume (Issue #1015 requested this; may or may not be shipped — check the actual binary's `--help`)
4. Agent-relay parses stdout for assistant text

#### Strategy C (last resort): Long-lived subprocess

1. Keep Crush running as a subprocess, pipe stdin/stdout
2. Less reliable, piping into interactive mode breaks TUI (Issue #2260 confirmed)
3. Only viable if server mode is too immature

**Cursor task: Before writing integration code, clone `charmbracelet/crush`, read `internal/server/`, `internal/client/`, and `internal/proto/`. Document the protocol, how to start the server, and how to send messages / receive events. This determines the entire agent-relay design.**

## crush run (non-interactive)

- Sends a single prompt, prints response to stdout, exits
- Prompt can be CLI args or piped from stdin via `MaybePrependStdin`
- `--yolo` flag auto-approves all tool calls (no user confirmation)
- `--model` flag selects model as `{provider}/{model}` format
- Creates a new session per invocation (`session_id` logged to stderr)
- Exit codes / output format need testing

## Session management

- Sessions stored in SQLite (`.crush/crush.db`)
- `crush session list --json` — lists sessions with JSON output (automation-friendly)
- Session resume has known edge cases with orphaned `tool_use` blocks (Issue #1206, partially fixed in v0.59.0)
- The `session.Service` is accessed via the Workspace interface

## Permission system

- `--yolo` (or `-y`), or `skip_permission_requests` in config) bypasses ALL permission prompts
- When `skip==true`, `Request()` returns true immediately — no UI interaction
- Can also be toggled at runtime via `SetSkipRequests(bool)`
- For Manifest: always run with `--yolo` or `skip_permission_requests: true` in config

## Config (crush.json)

Verified from DeepWiki + schema analysis:

**MCP key is `mcp`** (not `mcpServers`). Config struct at `internal/config/config.go:169-186`, schema at `schema.json:182-261`.

**Config merge order** (later wins):

1. Global: `~/.config/crush/crush.json`
2. Project: walk up from cwd looking for `crush.json` and `.crush.json`
3. Environment variables: `CRUSH_*` prefix

**Variable resolution:** String values resolved through `VariableResolver`

(`internal/config/load.go:97-98`). Test both `${VAR}` and `${{VAR}}` syntax against actual behavior.

**Correct crush.json for Manifest (DeepSeek + MCP):**

json

```
{
  "$schema": "https://charm.land/crush.json",
  "providers": {
    "deepseek": {
      "id": "deepseek",
      "name": "DeepSeek",
      "type": "openai-compat",
      "base_url": "https://api.deepseek.com/v1",
      "api_key": "$DEEPSEEK_API_KEY",
      "models": [
        {
          "id": "deepseek-v4-flash",
          "name": "DeepSeek V4 Flash"
        },
        {
          "id": "deepseek-v4-pro",
          "name": "DeepSeek V4 Pro"
        }
      ]
    }
  },
  "models": {
    "large": { "model": "deepseek-v4-pro", "provider": "deepseek" },
    "small": { "model": "deepseek-v4-flash", "provider": "deepseek" }
  },
  "mcp": {
    "browser": {
      "command": "npx",
      "args": ["-y", "@anthropic/mcp-playwright"]
    },
    "manifest-deploy": {
      "command": "/opt/manifest/mcp-docker-deploy"
    },
    "manifest-email": {
      "command": "/opt/manifest/mcp-email-send",
      "env": {
        "SMTP_HOST": "$MANIFEST_SMTP_HOST",
        "SMTP_USER": "$MANIFEST_SMTP_USER"
      }
    }
  },
  "options": {
    "skip_permission_requests": true,
  }
}
```

```
"disable_notifications": true,  
"disable_provider_auto_update": true,  
"initialize_as": "AGENTS.md"  
}  
}
```

Note: Use `openai-compat` type for DeepSeek (not `openai`). Crush README explicitly says: "`openai` should be used when proxying through OpenAI. `openai-compat` should be used when using non-OpenAI providers that have OpenAI-compatible APIs."

## Tools available to the agent

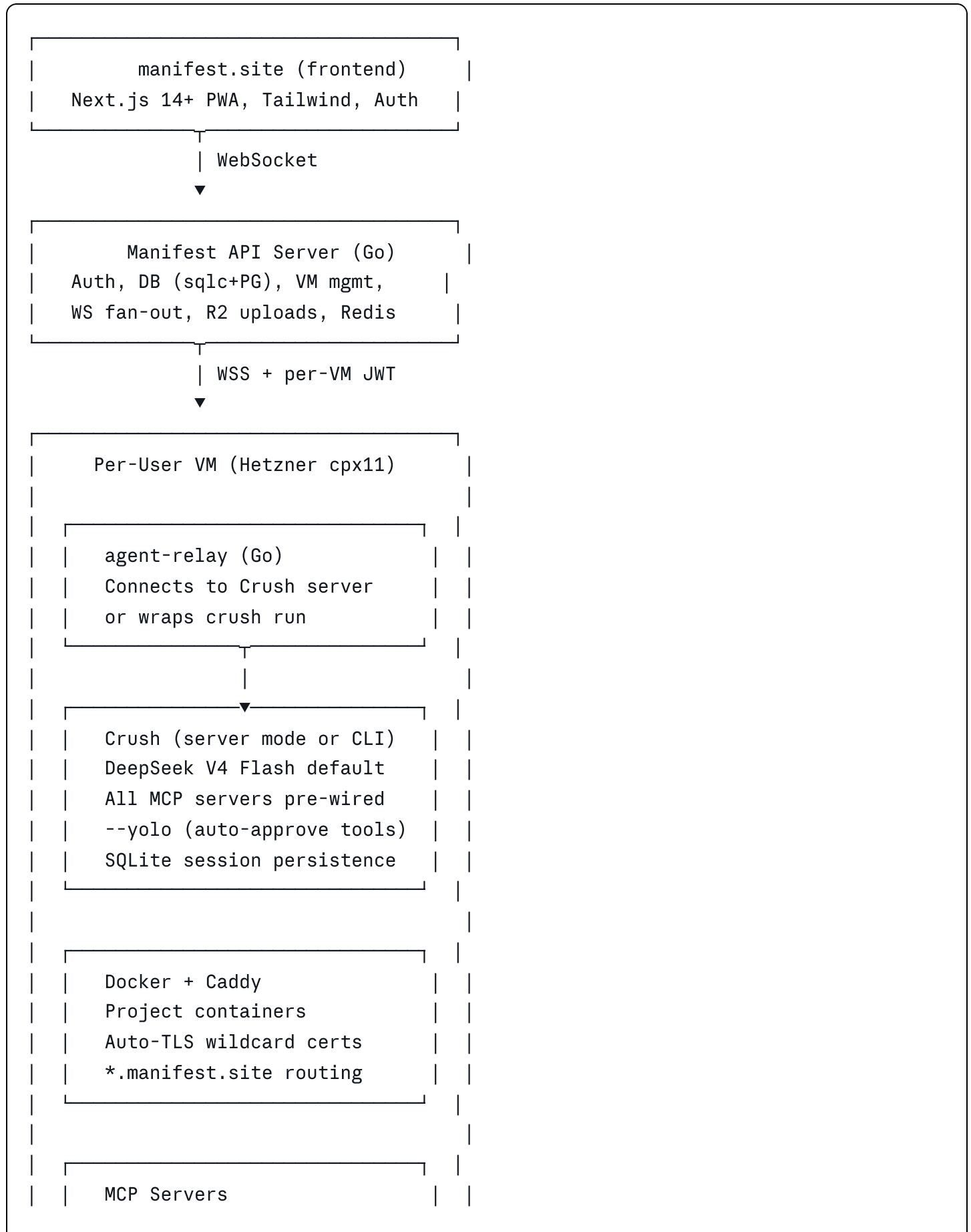
Built-in tools (from `internal/agent/tools/`):

- `bash` — shell execution
- `view` — read files with line ranges
- `edit` — targeted string replacement with diff
- `multiedit` — batch edits on a single file
- `write` — create/overwrite files
- `download` — fetch remote content to disk
- `ls` — list directory
- `grep` — search files
- `glob` — pattern matching
- `fetch` — HTTP fetch, converts HTML to markdown
- `agentic_fetch` — spawns sub-agent for web research
- `web_search` — DuckDuckGo integration
- `agent` — spawn sub-agents for complex tasks
- `crush_info` — agent reads its own config/status
- `crush_logs` — agent reads its own logs
- LSP tools: diagnostics, references
- MCP tools: whatever MCP servers are configured

## ACP (Agent Client Protocol) — future opportunity

Issue #2091: Someone already built a fork that uses Crush as an ACP client to drive Claude Code. ACP decouples agent UI from agent logic. This is exactly the Manifest use case. Worth watching — if Crush officially supports ACP server mode, the agent-relay becomes an ACP client connecting to Crush's ACP server.

## Architecture



		mcp-docker-deploy (custom)		
		mcp-email-send (custom)		
		mcp-twilio-sms (custom)		
		Playwright browser (npx)		
		Google Calendar (npx, OAuth)		
		Gmail read (npx, OAuth)		

## Implementation Phases

### Phase 0 — Crush server mode investigation (BLOCKING, do first)

Clone `charmbracelet/crush`. Read these files:

1. `internal/server/` — How does the server start? What does it listen on (TCP, Unix socket, HTTP)?
2. `internal/client/` — How does the client connect? What auth does it use?
3. `internal/proto/` — What's the wire protocol? Protobuf? JSON lines? What message types exist?
4. `internal/cmd/run.go:155-163` — The `CRUSH_CLIENT_SERVER` code path. How does `runNonInteractive` stream events from the server?
5. `internal/backend/` — How does this relate to `internal/server/`?

Document findings in `manifest/docs/crush-server-analysis.md`. This determines whether agent-relay is a Crush protocol client (Strategy A) or a subprocess wrapper (Strategy B/C).

Also check: `crush run --help` for `--session`, `--resume`, `--continue` flags. Any flag that resumes a previous session changes the multi-turn story.

### Phase 1 — VM base image (`manifest/vm-image/`)

Packer build: Ubuntu 24.04, Docker, Caddy, Node 20, Python 3.12, pinned Crush binary, Playwright/Chromium, custom MCP binaries in `/opt/manifest/`, systemd units.

`crush.json` template with `$(VAR)` expansion for user-specific secrets.

Exit criteria: `crush run --yolo "Create a file called hello.txt with the text hello world"` succeeds with DeepSeek V4 Flash on the VM.



## Phase 2 — Agent relay (manifest/agent-relay/)

If Strategy A (Crush server mode works):

- Start Crush server on VM boot via systemd
- Agent-relay is a Go WebSocket server that accepts connections from the API server
- Translates between Manifest's WebSocket protocol and Crush's native client protocol
- JWT auth: API server presents per-VM JWT on connect

If Strategy B (repeated crush run):

- Agent-relay spawns `crush run --yolo --session {session_id} "user message"` per message
- Captures stdout (assistant text) and stderr (tool call logs)
- Parses and relays structured events to the API server
- Session ID per project, stored in DB

Protocol between API and relay (regardless of strategy):

```
json

// API → Relay
{"type": "user_message", "session_id": "...", "content": "...", "attachments": [...]}

// Relay → API (streamed)
{"type": "text_delta", "content": "partial response..."}
{"type": "tool_call", "tool": "bash", "input": "echo hello"}
{"type": "tool_result", "tool": "bash", "output": "hello"}
{"type": "text_delta", "content": "more response..."}
{"type": "done", "session_id": "..."}
{"type": "error", "message": "..."}

```

## Phase 3 — mcp-docker-deploy (manifest/mcp-servers/mcp-docker-deploy/)

Custom MCP server (Go binary). Tools:

- `deploy_static(project_name, source_dir)` → builds Nginx container, configures Caddy reverse proxy, returns URL
- `deploy_app(project_name, dockerfile_dir)` → builds from Dockerfile, same flow
- `list_projects()` → running containers with URLs and status
- `stop_project(project_name)` / `restart_project(project_name)`

- `get_logs(project_name, lines)` → container logs for agent self-debugging

Port allocation: each project gets a port from a range (e.g. 10000-10100). Caddy config updated to route `{user}-{project}.manifest.site` to `localhost:{port}`.

Exit criteria: Crush builds a trivial `index.html`, calls deploy MCP, URL reachable.

## Phase 4 — API server (`manifest/api-server/`)

Go service. sqlc + PostgreSQL.

Data model:

- User (id, email, name, image, google\_token\_enc, tier, created\_at)
- VM (id, user\_id, hetzner\_id, ip, status, relay\_jwt\_hash, last\_active, created\_at)
- Project (id, user\_id, name, slug, subdomain, status, created\_at, updated\_at)
- Message (id, project\_id, role, content, attachments\_json, tool\_calls\_json, created\_at)
- ApiKey (id, user\_id, provider, key\_enc, created\_at)

Endpoints:

- NextAuth callbacks (Google + GitHub OAuth)
- `POST /api/projects` — create project, ensure VM is awake
- `GET /api/projects` — list user's projects
- `WS /api/ws/{project_id}` — WebSocket to chat. Authenticates user, resolves VM, opens relay connection, bidirectional relay.
- `POST /api/upload` — returns R2 presigned URL
- `PATCH /api/settings/keys` — store encrypted API keys, push to VM config

Hetzner lifecycle:

- Provision: create server from snapshot, inject cloud-init with user config + relay JWT
- Suspend: power off after idle (FREE: 5min, PRO: 30min)
- Wake: power on, wait for relay health check
- Cost model: compare suspended-disk-cost vs snapshot+delete+recreate. Decide per-tier.

## Phase 5 — DNS & routing

Wildcard DNS: `*.manifest.site` → load balancer. LB routes based on subdomain → VM IP (lookup from DB). On-VM: Caddy handles TLS + reverse proxy to Docker containers. Staging: single VM + `/etc/hosts`.

## Phase 6 — Frontend (manifest/frontend/)

Next.js 14+ App Router, Tailwind, PWA.

Routes:

- `/` — marketing landing page
- `/login` — OAuth
- `/app` — chat (new project)
- `/app/project/[id]` — project chat
- `/app/settings` — API keys, integrations, account

Components:

- ChatView, MessageBubble, InputBar (photos + links + text)
- BuildProgress (streaming tool calls rendered as steps)
- PreviewCard (thumbnail, URL, copy button when deploy completes)
- ProjectList, ProjectCard (status indicators, URLs)
- Sidebar (project list, user profile)

State: Zustand for client state, WebSocket for real-time.

## Phase 7 — MCP integrations & BYOK

Google Calendar/Gmail: OAuth in frontend → token stored encrypted in DB → pushed to VM env → Crush MCP servers pick it up.

Email send / SMS: platform Twilio + SMTP accounts. Per-user from address.

BYOK: settings page for DeepSeek/Anthropic/OpenAI keys. Encrypted in DB. Pushed to VM crush.json. Crush picks up new provider on next session.

## Phase 8 — Cost controls & tiers

FREE: 2 projects, 20 msgs/day, aggressive suspend (5min), platform DeepSeek key.

PRO (\$8/mo): unlimited projects, 500 msgs/day, 30min suspend, custom domains (stub), BYOK.

Message caps enforced in API layer before relay. Stripe integration stubbed.

## Phase 9 — Polish

Onboarding flow (VM provisioning spinner), error boundaries, never leak infra terms to UI, AGENTS.md rules for the model.

---

# AGENTS.md (system prompt for Crush on each VM)

markdown

## # Manifest Agent

You are Manifest, a friendly AI that builds web pages and tools for non-technical users

### ## Capabilities

- Build websites, photo galleries, schedules, landing pages, small apps
- Browse the web (fetch/web\_search tools + Playwright MCP)
- Read Google Calendar and Gmail (with user permission)
- Send emails from user's @manifest.site address
- Send SMS via Twilio
- Deploy anything as a live site at {user}-{project}.manifest.site

### ## Deploy workflow

1. Create project files in /home/manifest/projects/{project-name}/
2. Static sites: write index.html (CSS/JS inline or same dir)
3. Dynamic apps: write a Dockerfile
4. Call the manifest-deploy MCP tool
5. Return the live URL to the user

### ## Rules

- NEVER show code unless the user asks
- NEVER mention Docker, containers, Nginx, Caddy, ports, or infrastructure
- NEVER show error logs – say "Let me try a different approach"
- Keep responses conversational – you're a helpful friend
- Make photos look great by default (responsive grid, lightbox, clean typography)
- When given a URL, browse it and extract relevant data
- Default to clean HTML/CSS. Only use React/Vite if complexity demands it
- Clarify only when truly ambiguous – otherwise build the most reasonable interpretation

---

## Repository layout

```
manifest/
├── docs/
│   └── crush-server-analysis.md      # Phase 0 output
├── frontend/                       # Next.js 14+
├── api-server/                     # Go, sqlc+PG
└── agent-relay/                    # Go, bridges API↔Crush
```

```
|— vm-image/                                # Packer, cloud-init, systemd
|   |— config/
|   |   |— crush.json.template
|   |   |— AGENTS.md
|   |   |— Caddyfile.template
|   |   |— systemd/
|— mcp-servers/
|   |— mcp-docker-deploy/                  # Go
|   |— mcp-email-send/                    # Go
|   |— mcp-twilio-sms/                    # Go
|— docker-templates/
|   |— static-site/Dockerfile
|   |— node-app/Dockerfile
|   |— python-app/Dockerfile
|— docker-compose.yml                      # Local dev: PG + Redis + fake VM
|— README.md
```

---

## First milestone (vertical slice)

docker-compose with: Postgres, Redis, API server, one "VM" container (same base image as prod) running Crush with DeepSeek, agent-relay, and mcp-docker-deploy. A minimal Next.js page that sends one message over WebSocket, shows a streamed reply, and if the agent calls deploy, renders a preview card with a live URL on the compose network. Everything else hangs off that spine.