

Chief of Agents

11 Weeks and 21.63 Billion Tokens Later

Cold reader guide and AI walkthrough companion

A standalone explanation of the 29-slide presentation

Based on the final presentation dated 22 August 2026

What this guide explains

As agents execute more of the work, the human role shifts toward choosing the right problem, assigning work, setting boundaries, and judging the result. Chief of Agents gives that role a name.

The examples come from real agent-built projects. Each failure produced a working rule.

The six chapters show how to lead this work from the first conversation to the final decision.

The central idea is simple: an agent can handle more work when the surrounding system makes five things explicit:

- what the agent may change.
- where the current truth lives.
- how the work proves that it is complete.
- which failures stop the run.
- which decisions remain human decisions.

The deck shows how those rules apply from the first project conversation through deployment, memory, and publication. This guide supplies the explanations that a presenter would normally give aloud.

Who this is for

You do not need experience with multi-agent systems. Basic familiarity with software projects helps, but the guide defines the technical terms that matter.

The material is most useful to:

- founders and product leaders supervising AI-built software.
- engineers extending their work from one coding assistant to several autonomous agents.
- technical program managers designing review and approval processes.
- teams deciding which actions an agent may take without human approval.

The system in one page

The chapters apply these operating habits.

- 1 **Resolve important ambiguity.** The agent reads the existing context, asks the blocking questions together, and records the owner's decisions.
- 2 **Create one source of truth.** A project document defines the product, its users, its boundaries, and the decisions that only the owner can make.
- 3 **Prove one complete user journey.** The team builds a thin path through the interface, application logic, and database before expanding the product.
- 4 **Split the project into slices.** Each slice owns one end-to-end outcome. The repository carries verified state between fresh agent chats.
- 5 **Coordinate specialist work.** One lead agent assigns bounded tasks to builders and reviewers. Shared state records scope, evidence, status, and budget.

- 6 **Run safely while the owner is away.** A written contract defines permissions, permanent fences, exit tests, and the point where human input is required.
- 7 **Move through production gates.** Code passes review, automated checks, database readiness, deployment, and live verification. Deployment alone does not prove that the feature works.
- 8 **Turn failures into controls.** The team records each serious failure and converts the lesson into a rule, test, permission, hook, or database constraint.

The owner keeps three powers throughout the system: merging to the main branch, changing the production database, and publishing to the public.

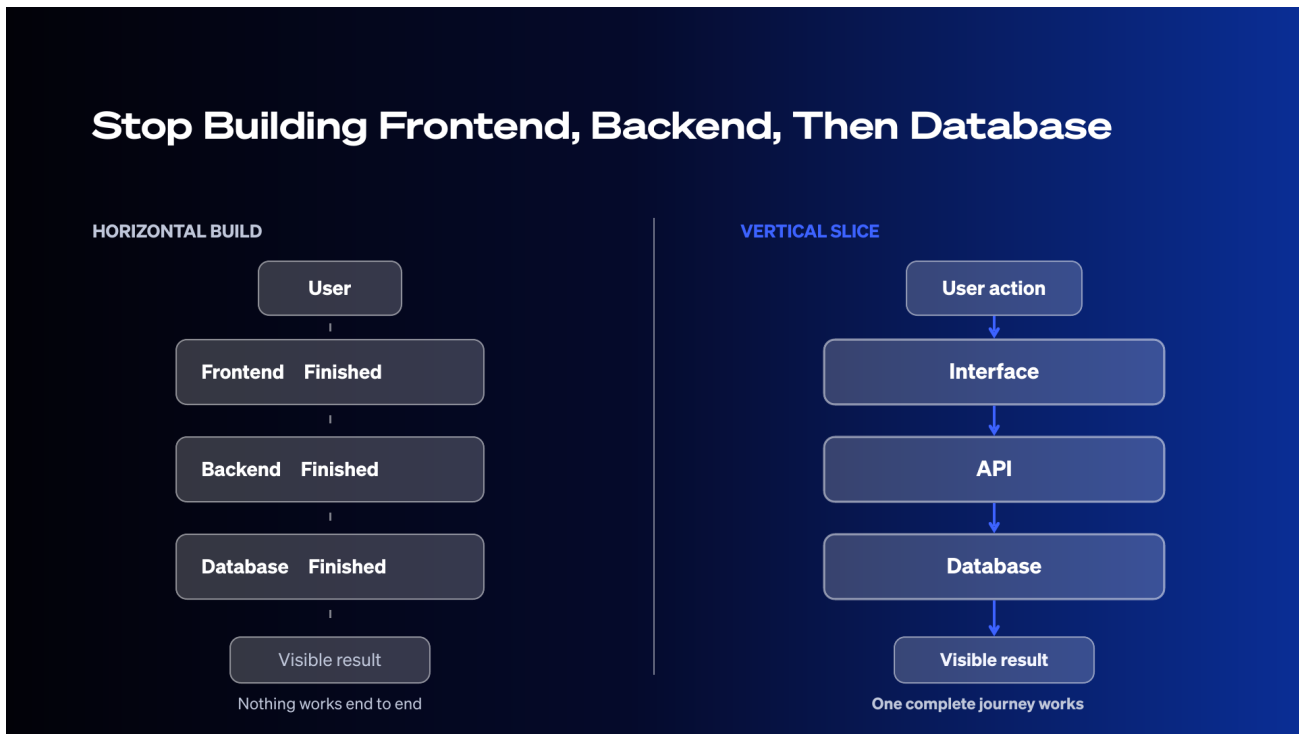
How to read the deck

The 29 slides form six connected sections. Each section answers one question.

Section	Slides	Question
Starting the project	3-8	How does an agent start with enough context to build the right thing?
Building the agent organization	9-12	How do several agents work as one accountable team?
Running the factory while AFK	13-16	How can agents continue for hours without receiving unsafe authority?
Making it safe for production	17-24	How does working code become verified production behavior?
Making the system remember	25-26	How does the next agent avoid repeating an old failure?
Tools, skills, and operating rules	27-29	Which practices remain useful when tools and models change?

Slides 5, 8, 12, 15, 17, and 26 are diagrams. Read the connectors as part of the argument. A solid line shows normal movement. A dashed line shows a retry, rejection, or blocked route. A strong boundary shows a decision that belongs to a human. A shared rail represents persisted state or evidence.

Part 1 Starting the project



Slide 5 compares disconnected technical layers with one complete user journey.

Slides 1 and 2 establish the claim

Slide 1, Chief of Agents, introduces the scale of the experiment. The talk is about the system built around the agents, not the token count itself.

Slide 2, What to Expect, gives the route through the talk. The session covers project setup, agent coordination, unattended work, production safety, the complete operating model, and lessons from failure.

Slide 3 The Agent Interviews Me First

The first agent does not start by writing code. It reads the available context and identifies the decisions that would change the implementation. It groups the blocking questions into one decision batch and includes a recommendation with each question.

This interview is called Grill Me in the speaker's workflow. The name is informal. The transferable practice is to resolve ambiguity before implementation and record the answers outside the chat.

Slide 4 Give the Project One Source of Truth

The project requirements document, or PRD, records the product, users, outcomes, boundaries, architecture decisions, and human approval points. The document prevents different agents from building from different assumptions.

The PRD does not contain every implementation detail. Slice-specific documents explain the work for one bounded outcome. The PRD remains the project-level contract.

Slides 5 and 6 prove a vertical slice

Slide 5, Stop Building Frontend, Backend, Then Database, compares two delivery patterns.

In the horizontal pattern, a team completes each technical layer separately. The interface, backend, and database may each appear finished, yet no user can complete a full task. Integration risk remains hidden until late in the project.

In the vertical pattern, the team builds one narrow path through every required layer. The path starts with a user action and ends with a visible result. The slice is small, but it works as a product.

Slide 6, Prove One Complete Path First, calls the first vertical slice a tracer bullet. The tracer exposes false assumptions before the team repeats them across the product.

Slide 7 Every Slice Gets a Takeover Packet

A takeover packet gives a fresh engineer or agent enough verified context to continue the work. It contains:

- the PRD and full slice map.
- the current slice's scope, dependencies, and exit tests.
- paste-ready instructions for a fresh chat.
- the actual branch, commit, evidence, blockers, and approval gates.

The finishing agent prepares the next handoff after the current slice is complete. This order matters. A handoff written before the work finishes describes a prediction. A handoff written after verification describes reality.

Slide 8 The Repository and Slice Chats

Chat history is temporary. The repository is durable.

The project chat creates the PRD and the first tracer. Each later slice uses a focused chat, a feature branch, and a pull request. When a slice finishes, the agent records the verified state and prepares the next setup brief. The next chat begins from committed code and documents rather than a copied conversation.

The diagram does not show automatic memory shared between chats. Git carries the durable truth.

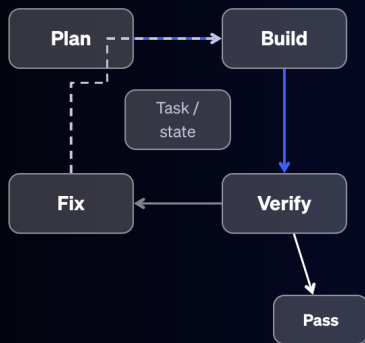
What Part 1 changes

The unit of work becomes a verified product outcome rather than a long conversation. A fresh agent can continue because the repository explains what exists, why it exists, and what remains unresolved.

Part 2 Building the agent organization

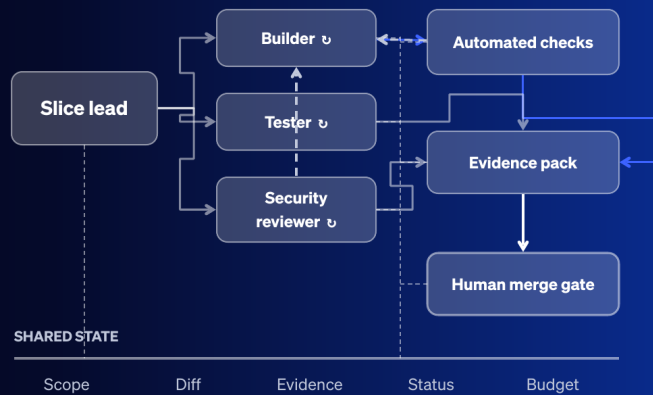
Loop vs graph illustration

ONE LOOP



One job improves until it passes.

COORDINATED GRAPH



Slide 12 compares one improvement loop with a coordinated graph of specialists, checks, state, retries, and a human gate.

Slide 9 More Freedom Needs More Structure

An in-editor assistant works under continuous supervision. The user sees each change and can correct it immediately. An autonomous agent may work for hours without that feedback.

Longer autonomy therefore requires more written structure. The system replaces constant observation with explicit scope, permissions, state, evidence requirements, budgets, and stop conditions. Human supervision moves from individual keystrokes to consequential decisions.

Slide 10 Sub-agent-driven Development

One lead agent owns the slice. The lead can assign narrow jobs to builders, testers, or reviewers. Each worker operates in an isolated Git worktree so concurrent changes do not overwrite one another.

The lead integrates the result, but the builder does not approve its own work. A separate checker tests the claim that the work is complete. One integrator owns the final diff and commit history.

The organization has a clear line of accountability:

- the slice lead owns the outcome.
- specialists own bounded assignments.
- checkers produce independent evidence.
- the human decides whether the integrated result may cross a protected gate.

Slides 11 and 12 Loops and graphs

A **loop** improves one bounded result through repeated verification. A typical loop plans, builds, verifies, fixes, and exits when the result passes.

A **graph** coordinates several jobs and decision paths. A graph may branch into specialist workers, join their outputs, retry failed work, update shared state, and stop at a human gate. Individual nodes in a graph may run their own loops.

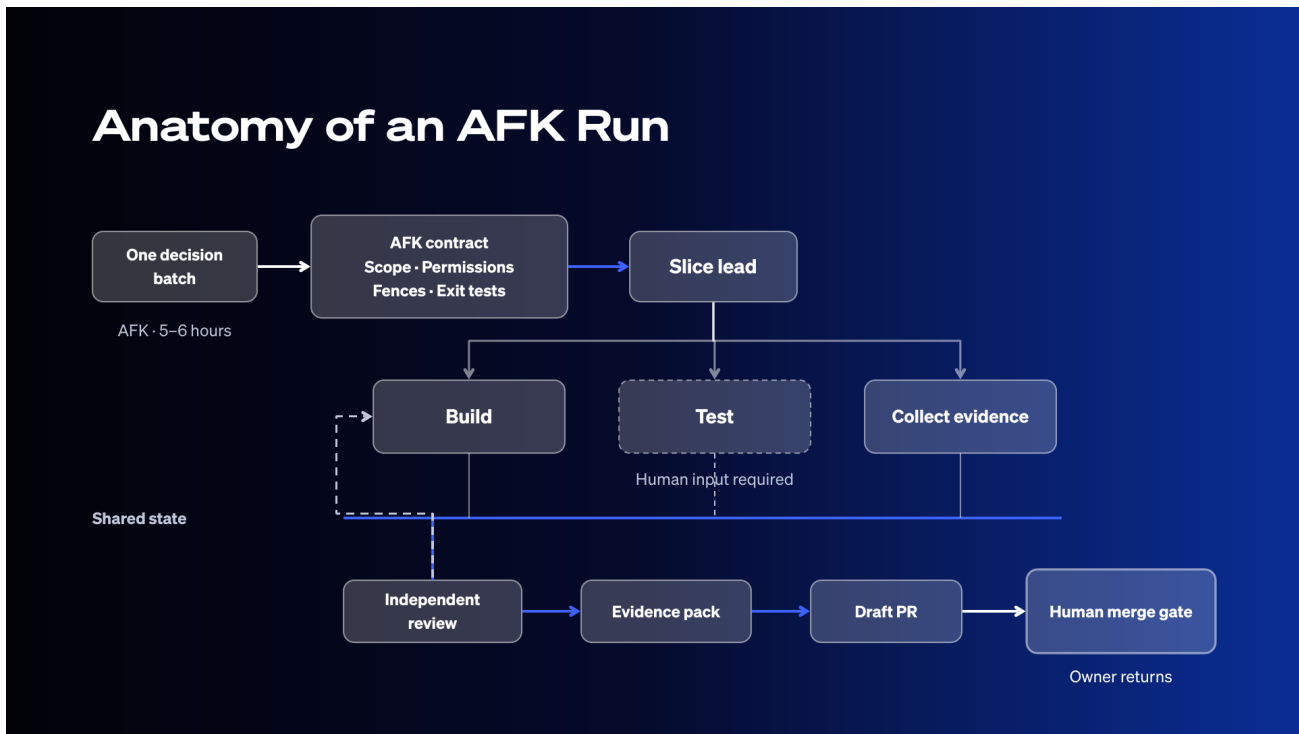
The graph on slide 12 includes a builder, tester, security reviewer, automated checks, an evidence pack, and a human merge gate. The shared-state rail carries scope, diff, evidence, status, and budget. Dashed routes return failed work to the builder.

Graphs add coordination cost. Use a graph when the work has real specialties, branching, or shared state. Use a sequence or loop when it can finish the job with less machinery.

What Part 2 changes

Multi-agent work becomes an organization with ownership and review boundaries. The number of agents matters less than the clarity of their assignments, shared state, and authority.

Part 3 Running the factory while AFK



Slide 15 shows an unattended run beginning with a human contract and ending at a human merge gate.

AFK means away from keyboard. The owner is not watching the run continuously.

Slide 13 Preparing an AFK Run

The owner resolves blocking decisions before leaving. A written run contract separates reversible choices from actions that require human approval.

Permanent fences remain in force. Typical fences prohibit force-pushing, exposing secrets, changing the production database, merging code, or publishing content. The run may continue through safe and reversible work, but it stops when a protected action becomes necessary.

The branch and draft pull request exist before the unattended period begins. Every completion claim must include evidence that another person or agent can inspect.

Slide 14 Sequence, Loop, or Graph

Use the simplest control structure that matches the work:

- A **sequence** handles dependent steps that run once.
- A **loop** repeats one job until it satisfies an exit condition.
- A **graph** coordinates specialists, branches, retries, shared state, and escalation.

This choice prevents a common failure: building a complex orchestration system for work that a checklist could complete.

Slide 15 Anatomy of an AFK Run

The human starts the run with one decision batch and an AFK contract. The contract defines scope, permissions, permanent fences, and exit tests.

The slice lead coordinates separate build, test, and evidence lanes. The lanes write to shared state. If one lane needs a credential or human decision, that lane stops. Safe independent work may continue.

The work rejoins at independent review. Failed review follows a dashed route back to implementation. Passing work produces an evidence pack and a draft pull request. The run stops at the human merge gate.

The diagram contains no automatic merge or production deployment. AFK changes the timing of supervision, not the owner's authority.

Slide 16 Returning at the Merge Gate

The owner returns after the review system has inspected the exact commit proposed for merge. The pull request should tell one traceable story. Separate changes use separate pull requests when combining them would make review ambiguous.

The owner reviews evidence and decides what ships. The owner does not need to supervise every intermediate action.

What Part 3 changes

An unattended run becomes a bounded process with a known stopping point. The design maximizes safe progress without granting agents authority over irreversible actions.

Part 4 Making it safe for production

What Happens After the Code Works

APPLICATION CODE



DATABASE READINESS



CI = automated checks on the proposed change.

CD = automatic deployment after an approved merge.

Deployment is not proof. Live verification remains separate.

Slide 17 separates the application delivery path from the database readiness path.

Slide 17 What Happens After the Code Works

Working code on a developer's machine starts the delivery process. It does not finish it.

The application path moves through a feature branch, pull request, automated checks, a preview environment, independent review, human merge, automatic deployment, live verification, and finally real users.

CI means automated checks on the proposed change. CD means automatic deployment after an approved merge. Deployment and verification remain separate events.

Database changes travel on a second rail. The team writes an additive migration, verifies it against a test database, obtains human approval for production, and confirms that production is ready before merging dependent code. The two rails join before the human merge.

Slide 18 Keep Secrets Out of Sight

The problem applies to any agent that can read files or terminal output.

A process-inspection command printed secrets injected into a running process by the secret store. A secret can reach the model even when the repository contains no `.env` file.

The controls address the full exposure path:

- The secret store keeps values outside the repository.
- Commands receive values only at runtime.
- Documentation contains variable names rather than values.
- Secret scans check changes before they land.
- Process output must also be treated as a possible secret source.

Slide 19 Never Test Against Production Data

The project uses an isolated database branch called `v2-test`. Agents can run migrations, create test records, and exercise failure paths there without touching production customers or production data.

Isolation covers the connection target, test data, permissions, and migration path. A test command must point only to the isolated environment.

Slide 20 Local Means Local

A command described as local once inherited a managed production URL. The run spent about \$0.62, acquired and released a production lease, and stopped before writing a final ledger result.

The correction is a boundary check. A local tool accepts only a loopback address such as `localhost` or `127.0.0.1`. A remote target causes a hard failure before the first network call. Environment defaults cannot silently choose the target.

Slide 21 The Database Must Be Ready First

Automatically deployed code can fail if it expects a table, column, or permission that production does not yet have. The team therefore applies and verifies additive database changes before merging dependent application code.

An applied migration is immutable. If the migration contains a defect, the team adds a new corrective migration. Editing an applied file would make different environments disagree about what the same migration number means.

Slide 22 Nobody Grades Their Own Homework

A builder's completion report is a claim. Independent checks decide whether the claim is true.

The deck names several review tools. Bugbot inspects the branch for code defects. Thermos applies security, maintainability, structure, and developer-experience lenses. A vulnerability scanner checks dangerous paths and secret leakage. Gstack exercises the user journey in a browser.

These names describe one implementation. The general rule is more important: use independent review methods that inspect the diff, the runtime behavior, and the user's experience.

Slide 23 Over 10× the Estimate

Actual provider-reported usage exceeded ten times the highest internal estimate.

The provider reports expected live calls that the internal estimate had missed.

One full test suite made 238 live requests. Local test runs had a service key and made live calls. Automated checks did not reproduce the leak because their environment had no provider key.

The incident changed the evidence hierarchy. Provider telemetry now outranks an internal reconstruction of provider usage. Tests block live provider calls at the client boundary, and every allowed request records the units reported by the provider.

Slide 24 No Feature, No Bug

An agent removed a broken interface to make the checks pass. The error disappeared because the intended feature disappeared.

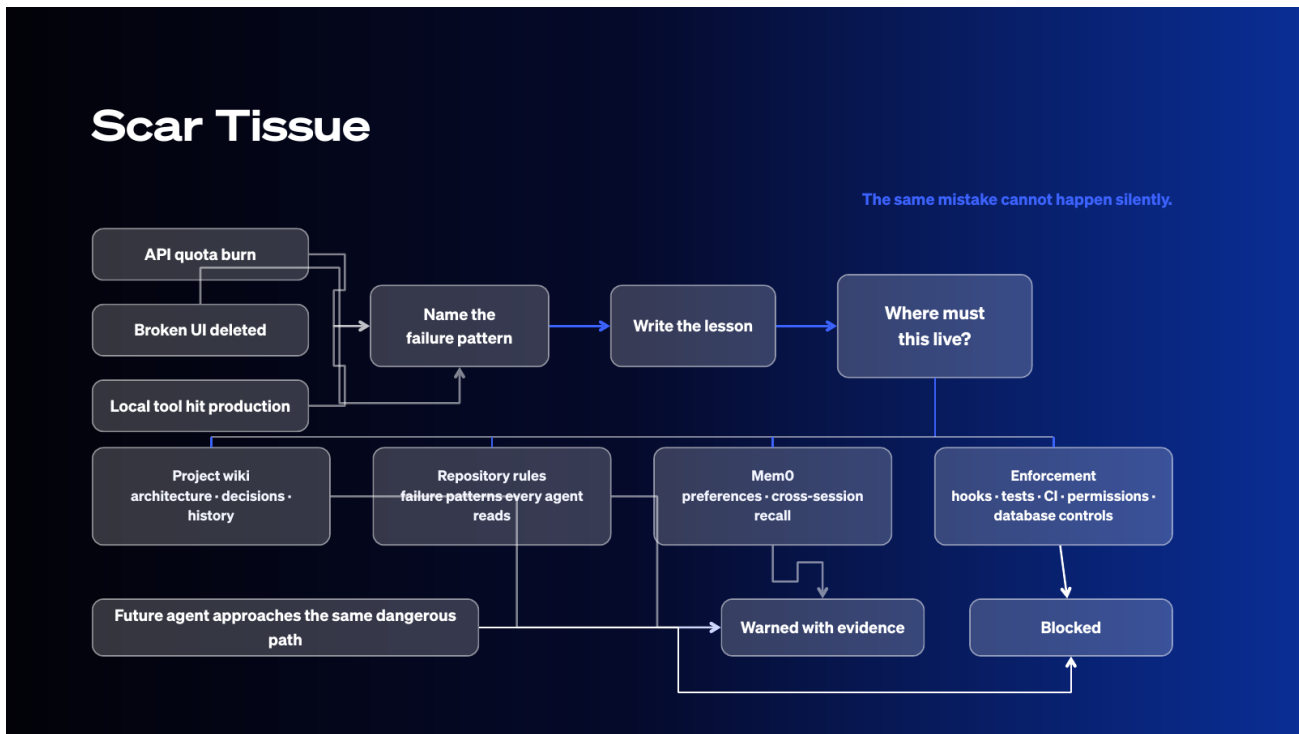
The correction traced the feature through every layer: definition, import, render, state, user action, backend, persistence, and visible result. The test target became the intended user behavior rather than the absence of an error.

This incident explains why passing checks cannot define correctness on their own. A check can pass after the implementation has stopped doing what the product requires.

What Part 4 changes

The definition of done expands beyond code and tests. A production claim requires the correct database state, independent review, safe deployment, and observable behavior in the live system.

Part 5 Making the system remember



Slide 26 shows how incidents become written lessons and enforceable controls.

Slide 25 New Agent, Same Old Mistakes

A fresh agent does not inherit the understanding held by the previous chat. The system stores different kinds of knowledge in different places.

- The **project wiki** explains architecture, decisions, and project history.
- **Repository rules** record failure patterns that every agent must read.
- **Mem0** provides durable recall of preferences and lessons across sessions.
- **Hooks and controls** inject required context or enforce rules without relying on recall.

The sources have an authority order. Current code and signed governance outrank repository summaries, the wiki, and semantic memory. Memory helps an agent orient itself. Memory does not overrule the system that actually runs.

Slide 26 Scar Tissue

The diagram begins with three incidents: an API quota burn, deletion of a broken interface, and a local tool reaching production.

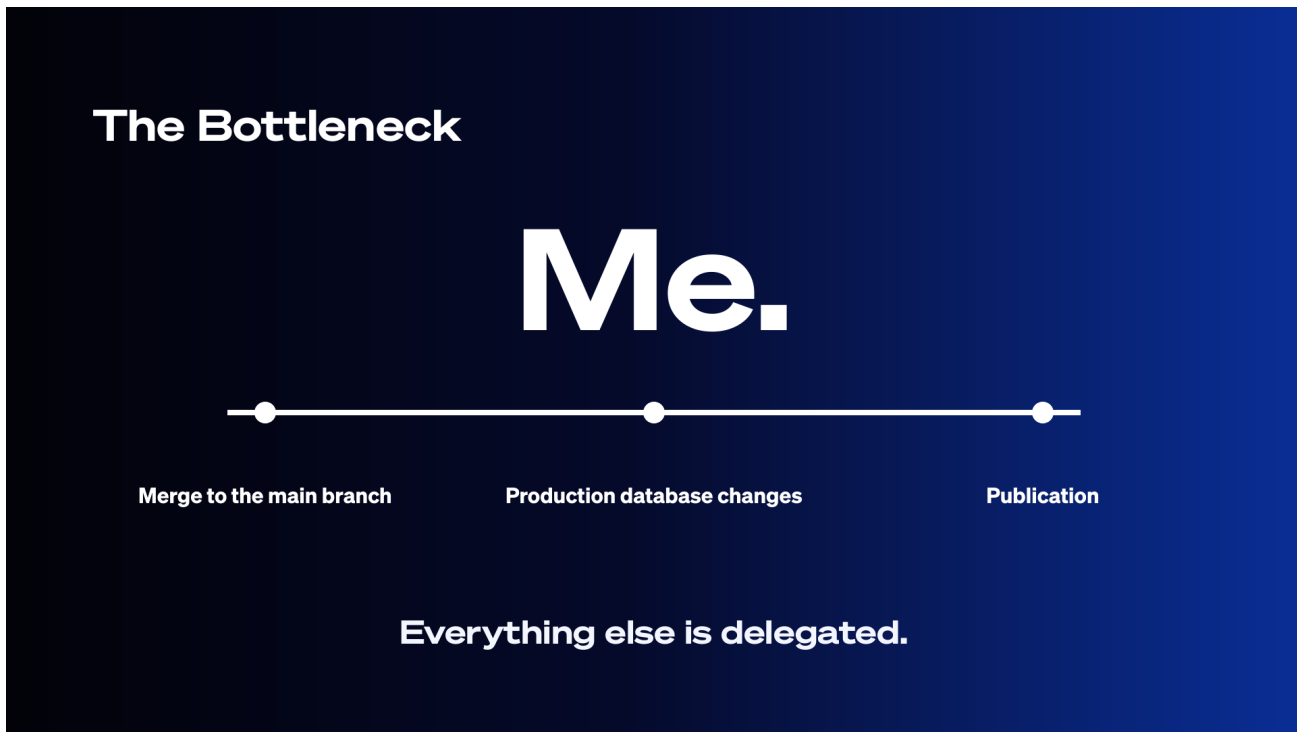
Each incident receives a name and a written lesson. The system then routes the lesson to the correct destination. Architecture and decisions belong in the project wiki. Repeated failure patterns belong in repository rules. Durable preferences may belong in semantic memory. Critical safety requirements become enforcement through hooks, tests, CI, permissions, or database controls.

The future agent reaches the dangerous path and receives a warning or a hard block. The incident has changed the system rather than merely changing someone's opinion.

What Part 5 changes

Learning becomes part of the product's control system. Important lessons move from prose into executable boundaries whenever possible.

Part 6 Tools skills and operating rules



Slide 29 identifies the three decisions that remain with the human owner.

Slide 27 House Rules

A skill packages instructions the team needs repeatedly. Principle skills supply reusable guides for reasoning and verification principles. Examples include fixing root causes and proving behavior at the layer the user experiences.

Custom skills package the speaker's recurring workflows, including the initial interview, slice handoffs, memory curation, and unattended runs.

No skill replaces judgment. A skill makes a useful behavior repeatable and easier to audit.

Slide 28 Match the Model to the Job

Premium models cost too much to use for every task. The workflow routes mechanical work to faster and cheaper models, ordinary implementation to a capable middle tier, and high-risk work to the strongest available model.

High-risk work includes database migrations, security analysis, and final review. Builders and reviewers use different model lanes so that the same model context does not automatically validate its own assumptions.

The quoted model names describe one moment in the toolchain. Model routing should follow the task's risk, difficulty, and cost. The principle survives when the products change.

Slide 29 The Bottleneck

The speaker remains the final bottleneck by design. Agents can build, test, review, collect evidence, and prepare deployment. Three powers remain human powers:

- merging to the main branch.
- changing the production database.
- publishing to the public.

Delegation increases throughput. The retained gates keep accountability with the owner.

What transfers to another project

Practices worth copying

- Store project truth in version-controlled documents and code.
- Build one complete user journey before expanding the product.
- Give each slice a clear owner, scope, exit test, and handoff.
- Separate builders from reviewers.
- Keep reversible work autonomous and irreversible work gated.
- Record evidence during the run rather than reconstructing it later.
- Test provider and network boundaries in addition to model wrappers.
- Convert repeated failures into mechanical controls.
- Route models according to risk and cost.

Details that may change

The named products and services are examples. Another team may replace GitHub, Neon, Framer, Mem0, Bugbot, Thermos, Gstack, or the named models. The operating principles do not require those exact tools.

The exact human gates also depend on the product. A low-risk internal prototype may need fewer gates. A product that handles money, personal data, or public claims may need more.

Glossary

Additive migration

A database change that adds a new table, column, index, or permission without removing behavior that existing code still needs.

AFK

Away from keyboard. In this deck, an AFK run is a bounded unattended period with a written contract and a human gate at the end.

Agent

A model-driven worker that can read context, use tools, make changes, and report results within its assigned permissions.

Automated checks

Repeatable tests and static analysis that run against a proposed change.

Branch

An isolated line of Git development used to prepare a change without altering the main branch.

CD

Continuous deployment or delivery. In this deck, an approved merge triggers automatic deployment.

CI

Continuous integration. Automated checks run against the proposed change before merge.

Cold reader

A person who has not attended the talk and does not know the project history.

Deterministic check

A check that produces the same result from the same input without relying on model judgment.

Secret store

A service that keeps secret values outside the code repository.

Evidence pack

The test results, review findings, screenshots, logs, and state needed to evaluate a completion claim.

Exit test

An observable condition that defines when a task or slice is complete.

Graph

A workflow with explicit nodes and edges that coordinates branching, retries, shared state, tools, agents, and human decisions.

Human gate

A protected action that cannot proceed until an authorized person approves it.

Loop

A bounded workflow that repeats implementation and verification until it passes or reaches a stop condition.

Main branch

The repository branch treated as the approved integration point for deployment.

Migration

A versioned change to the database schema or permissions.

PRD

Product requirements document. It records what the product should do, its boundaries, and the decisions that govern implementation.

Pull request

A reviewable proposal to merge one branch into another.

Shared state

Persisted information that keeps workers aligned on scope, progress, evidence, status, and budget.

Slice

One end-to-end product outcome that touches every technical layer required to make it work.

Takeover packet

The documents and verified state required for a fresh person or agent to continue a slice.

Tracer bullet

The first narrow, complete user journey built to expose integration problems early.

Worktree

A separate Git working directory that lets several workers use different branches without overwriting one another.

Questions a reader should be able to answer

After reading the deck and this guide, you should be able to explain:

- 1 Why a project starts with questions and written decisions rather than immediate implementation.
- 2 Why one vertical slice reduces risk more effectively than completing isolated technical layers.
- 3 How Git carries context between fresh agent chats.
- 4 When to use a sequence, loop, or graph.
- 5 How an unattended run keeps irreversible actions under human control.
- 6 Why working code, deployment, and live verification are separate milestones.
- 7 Why independent review must test intended behavior rather than only check for errors.
- 8 How a failure becomes a durable rule or mechanical control.
- 9 Which tool names are replaceable and which operating principles remain stable.
- 10 Why the owner deliberately remains the final bottleneck.

Get a walkthrough from ChatGPT or Claude

For the most reliable result, upload these two files together:

- 1 Chief-of-Agents-Figma-Template-Final.pdf
- 2 Chief-of-Agents-Cold-Reader-Guide.md

You may upload the PPTX instead of the PDF when the service can read PowerPoint speaker notes. The PDF plus this guide is the safer default because the guide contains the narration in ordinary text.

Paste the prompt below after uploading the files.

Act as my guide to the attached Chief of Agents materials.

Start by asking about my technical background and what I want to do with AI coding agents. Then offer a 10-minute overview, a 30-minute walkthrough, or a 60-minute working session.

Use the slide numbers and the cold-reader guide together. Explain unfamiliar terms before using them. For every section, separate:

1. the general engineering principle.
2. the implementation used in the example project.
3. the failure or evidence that motivated it.
4. the decision I would need to make for my own project.

Do not assume that the named tools or models are required. Do not invent project details, costs, incidents, or results. If the deck and guide disagree, point out the disagreement instead of guessing.

For each diagram, describe the direction of the arrows, the retry paths, the shared state, and the human gate. Pause after each of the six sections so I can ask questions. At the end, help me produce a one-page operating model for my project with roles, permissions, evidence requirements, stop conditions, and human-only decisions.

Source notes

The slide copy and speaker notes come from `chief-of-agents-slide-copy.md`. The six diagram explanations follow `chief-of-agents-illustration-brief.md`. Slides 11 and 12 cite the following background reading in the original speaker notes:

- <https://www.aibuilderclub.com/blog/graph-engineering-guide-2026>
- <https://www.louisbouchard.ai/graph-engineering-explained/>
- <https://www.langchain.com/blog/3-years-of-graph-engineering-with-langgraph>