CLAUDE CODE ULTIMATE GUIDE

Claude Code: From Zero to Productive

The fundamentals to get started effectively

Florian BRUNIAUX

October 2026

v1.5.4

Guide v3.44.1

Table of contents

1 What is Claude Code 4
1.1 Installation 5
2 Quickstart: Your first session 7
2.1 Launch Claude Code 7
2.2 The workflow 7
2.3 Concrete example 7
2.4 Essential commands 8
2.5 Keyboard shortcuts 8
3 The 5 golden rules 10
3.1 The key mindset: Context system, not chatbot 10
3.2 The WHAT/WHERE/HOW/VERIFY formula 11
3.3 Context management 11
4 The whitepaper series 13
4.1 #1: Prompts That Work (~15 min) 13
4.2 #2: Customizing Claude (~20 min) 13
4.3 #3: Security in Production (~25 min) 14
4.4 #4: Architecture Demystified (~20 min) 14
4.5 #5: Deploying Across Teams (~20 min) 14
4.6 #6: Privacy & Compliance (~15 min) 15
4.7 #7: The Reference Guide (~2h) 15
5 Recommended paths 17
5.1 Solo vs team vs enterprise 17
6 Resources 19
6.1 Reliability conventions 19
7 The series 20

📌 **Who is this whitepaper for?**
Primary audience: Everyone (developers, tech leads, managers)

Prerequisites: None (series entry point)

Reading time: 15 min

Key takeaways:

- Claude Code = autonomous agent with tool access, not a chatbot
- The WHAT/WHERE/HOW/VERIFY formula structures every effective interaction
- CLAUDE.md is the project's persistent memory; create it first

In 5 lines: Claude Code is not a chat assistant with superpowers. It's an autonomous agent that reads your files, executes your commands, and modifies your code directly. The structural difference from previous tools (Copilot, ChatGPT) is that Claude Code acts; it doesn't suggest. This guide explains how to turn that potential into measurable productivity, from first session to team-wide adoption.

1 What is Claude Code

Most developers who adopt Claude Code make the same mistake in the first weeks: they use it like a fancy chatbot, with vague prompts, and wonder why results are mediocre. This guide exists to prevent that.

Claude Code is Anthropic's official CLI tool for AI-assisted development. In one sentence: **an autonomous coding agent that operates directly in your terminal, reads your codebase, proposes changes, and executes commands with your explicit approval**.

> **After this introduction, you will know**: how to install Claude Code, launch your first session, apply the 5 golden rules, and choose your path through the series.

Feature	Description
Native CLI	No imposed IDE, works in any terminal
Agentic	Analyzes, plans, executes - not just autocompletion
Extensible	MCP (Model Context Protocol) to connect external tools
Customizable	Agents, Skills, Commands, Hooks to adapt to your workflows

Since late 2025, several agentic capabilities have become available and significantly change what Claude Code can do autonomously:

Feature	Since	Description
Tasks API	v2.1.16	Persistent task lists with dependencies; opt in on newer models with `CLAUDE_CODE_ENABLE_TODO_TOOLS=1`
Auto-memories	v2.1.32	Automatic cross-session context capture in `~/.claude/projects/`
Agent Teams	v2.1.32	Multi-agents in parallel via `TeamCreate`/`SendMessage`
LSP Tool	v2.0.74	Diagnostics after edits and symbol navigation, enabled by an official code intelligence plugin per language
Remote Control	v2.1.51	Control a local session from mobile or browser (Pro/Max/Team/Enterprise)
MCP Elicitation	v2.1.76	MCP servers can request structured data during a task
1M token context	v2.1.75	Native on current Opus, Sonnet, Haiku and Fable; older models have plan/provider restrictions
`/loop`	v2.1.71	Recurring automation at a fixed interval (e.g. `/loop 5m /qa`)
`/branch`	v2.1.78+	Parallel session forking (`--fork-session`) to explore multiple approaches
`--bare`	v2.1.78+	Scripted mode without hooks/LSP/plugins, ideal for CI/CD
Auto mode (Max)	v2.1.100+	Automatic permission decisions by a classifier, subject to account and policy availability

Feature	Since	Description
`/ultrareview`	v2.1.114+	Cloud-based parallel multi-agent code review (official skill); also available in CI/CD via `claude ultrareview` (v2.1.120)
`/goal`	v2.1.139+	Autonomous completion loop: set a condition, Claude keeps working until a separate evaluator (Haiku by default on the Claude API) checks it's met, no "continue" prompts needed
`xhigh` **effort (Opus 4.7)**	v2.1.100+	Additional effort tier between `high` and `max`; also supported by current adaptive models
`--safe-mode`	v2.1.169	Troubleshooting flag: disables all customizations (CLAUDE.md, plugins, skills, hooks, MCP servers)
Opus 5.5	v2.1.280	Current direct-service default, native 1M context; standard $4/$20 and fast $8/$40 per Mtok
Fable family	v2.1.170	Fable 5.1 is the current direct-API model where available; provider aliases differ
Nested sub-agents	v2.1.219	Agents can delegate to sub-agents up to 3 levels deep by default
Sonnet 5.5	v2.1.284	Current direct-API Sonnet, native 1M context; standard $2/$10 per Mtok. Sonnet 5 is legacy

Set nested delegation depth with `CLAUDE_CODE_MAX_SUBAGENT_SPAWN_DEPTH`.

1.1 Installation

Two installation options. The native installer is the method recommended by Anthropic; npm remains functional if you have specific constraints.

```
# Recommended method: native installer (no Node.js dependency)
curl -fsSL https://claude.ai/install.sh | bash

# Alternative: via npm (not recommended, but still functional)
npm install -g @anthropic-ai/claude-code

# Verify installation
claude --version

# First authentication
claude
# Follow the instructions to link your Anthropic account
```

Prerequisites: An eligible Claude subscription or API/provider access. Node.js 18+ is needed only for the npm installation path.

Go further: §1.1 Full Installation: troubleshooting, detailed prerequisites, initial configuration

2 Quickstart: Your first session

2.1 Launch Claude Code

Always launch Claude Code from the root of the project you are working on. This starting directory determines which files are analyzed and which `CLAUDE.md` is loaded into context.

```
cd my-project
claude
```

Claude automatically analyzes your project and waits for your instructions.

2.2 The workflow

The interaction model supports a human approval cycle. Whether an action prompts depends on permission mode, saved allow rules, and organization policy. Read operations and previously allowed actions can run without another prompt.

```
You describe -> Claude analyzes -> Claude proposes -> You approve -> Claude executes
```

Key principle: Choose the permission mode deliberately and review the resulting changes before shipping.

2.3 Concrete example

Here is what a typical interaction looks like. Notice how Claude breaks the task into visible steps and waits for your approval before applying any change.

```
> Add email validation to the login form

Claude will:
1. Read src/components/LoginForm.tsx
2. Propose a change with diff
```

3. Wait for your "y" to apply
4. Optionally run tests

2.4 Essential commands

These commands cover the vast majority of daily interactions. The first three (`claude`, `/status`, `/compact`) are the minimum set to learn on day one; the rest become second nature within a week of regular use.

```
claude              # Launch Claude Code
/init               # Generate an initial CLAUDE.md for your project
/status             # Check session and account state
/compact            # Compress context (at 70%+)
/usage              # Unified costs, stats, and session graphs (v2.1.118, replaces /cost and /stats)
/context            # Inspect token usage; use @file in a prompt to reference a file
/clear              # Start fresh
/plan               # Read-only mode (safe exploration)
/doctor             # Diagnose config (sanity check)
```

2.5 Keyboard shortcuts

Keyboard shortcuts let you switch between Claude Code modes without leaving the terminal. The most frequently used is `Shift+Tab`, which cycles between normal mode, auto-accept (for trusted changes), and plan mode (read-only exploration).

Shortcut	Action
`Shift+Tab`	Cycle: default -> auto-accept -> plan mode
`Alt+T`	Toggle thinking where supported; no effect on Opus 5.5, Sonnet 5.5, Haiku 5.5 or Fable
`Ctrl+B`	Run command in background
`Ctrl+R`	Command history
`@file`	Reference a specific file
`!command`	Execute a shell command
`Ctrl+C`	Cancel current operation
`Esc` x2	Open rewind and choose a checkpoint

📌 Pitfall to avoid: Permission Fatigue

The #1 friction point reported by the community: approving permission prompts without reading them, then reaching for `--dangerously-skip-permissions` on a non-sandboxed machine. A malformed command or prompt injection can then execute code without any control.

Choose the right mode for each context:

Mode	When to use
Default (explicit approval)	Daily development, new projects
Auto-accept (Shift+Tab)	Trusted scripts, controlled CI/CD
bypassPermissions	Isolated sandboxes only
`--dangerously-skip-permissions`	Avoid: high risk without a sandbox

Simple rule: if you are not in an isolated sandbox, keep manual approval on.

3 The 5 golden rules

1. **Always review diffs** before accepting changes
2. **Use** `/compact` at a useful task boundary; inspect `/context` (see reminders below)
3. **Be specific** in your requests (see formula below)
4. **Plan Mode first** for complex or risky tasks
5. **Create a CLAUDE.md** for every project

3.1 The key mindset: Context system, not chatbot

> **"Stop treating it like a chatbot. Give it structured context. Changes everything."**
> (Robin Lorenz)

Claude Code is not a fancy chatbot; it's a **4-layer context system**. Each layer represents a level of persistent memory that you build progressively: the first (CLAUDE.md) takes minutes to set up, while the others are added over weeks as you identify recurring patterns in your workflow.

Layer	Mechanism	Setup
Memory	`CLAUDE.md`	Week 1
Knowledge	Skills	Week 2
Automation	Hooks	Week 3
Persistence	Project memory	Ongoing

Result: each layer makes the next one more effective.

> **Auto Dream** (v2.1.78+): after 24h of inactivity and 5 sessions, Claude automatically consolidates its project memory in 4 phases (orient, gather signal, consolidate, prune and index). Accessible via `/memory`.

> 📌 **Illustrative scenario: Antoine, full-stack dev at a tech consultancy**
> Antoine starts with a basic `CLAUDE.md` listing his stack (Next.js, Prisma, PostgreSQL). In the first week, he notices Claude follows his conventions without needing reminders. In week 2, he adds a "REST API conventions" Skill that encodes his validation patterns. In week 3, a pre-commit hook automatically checks that Prisma migrations are generated. Result: his prompts go from 4 lines of context to a single instruction, and Claude produces compliant code on the first try.

3.2 The WHAT/WHERE/HOW/VERIFY formula

To define the deliverable, scope, approach and verification:

Element	Description
WHAT	Concrete expected deliverable
WHERE	Paths to relevant files
HOW	Technical constraints, desired approach
VERIFY	Measurable success criteria

Example:

> Add input validation to login form.
> WHERE: src/components/LoginForm.tsx
> HOW: Use Zod, inline errors
> VERIFY: Empty email shows error message

3.3 Context management

Context is Claude's working memory. Current Opus, Sonnet, Haiku and Fable models have native 1M windows on the direct API; Haiku 4.5, the last 200K model, is legacy. Provider settings and older model variants can differ. Each teammate has an independent window sized by its model. Inspect `/context`:

Zone	Context	Action
Green	0-50%	Work freely
Yellow	50-70%	Be selective
Orange	70-90%	Consider `/compact` at a task boundary
Red	90%+	Save decisions, then compact or prepare a handoff

> 📌 **Context and billing limits**
> The March 2026 peak-hour announcement is historical, not a current quota guarantee. Check `/usage` and the active plan terms before scheduling a heavy task. Separate the context window, subscription limits and API billing. Subscription and billing reference.

> 📌 **Memo: Manage your context in 3 reflexes**
> **What**: Keep the context relevant to the active task. Percentage thresholds are reminders, not guarantees of recall.

When: Check with `/status` after each major task, or when responses become vague or repetitive.

How: (1) `/compact` at a useful task boundary; (2) Start a new session for each independent task; (3) Use `@file` to load only relevant files instead of letting Claude scan everything.

Verify: If Claude loses a constraint or produces an unsupported answer, reopen the relevant evidence. Context percentage alone does not identify the cause.

Go further: TL;DR: The 5-Minute Summary: complete visual summary with examples

4 The whitepaper series

The foundations are set: you know how to install Claude Code, structure your requests, and manage your context. Each whitepaper that follows dives deeper into a specific aspect, from writing effective prompts to multi-agent orchestration. You don't need to read them all: the paths table at the end of this section will guide you based on your profile.

This introduction is part of a series of 13 technical whitepapers.

4.1 #1: Prompts That Work (~15 min)

→ See **#1 Prompts That Work** (`01-effective-prompts.qmd`)

Audience: Developers of all levels.

- Apply the WHAT/WHERE/HOW/VERIFY formula to define a result you can verify
- Structure your prompts so Claude understands context without re-reading
- Identify anti-patterns (vague prompts, overly broad requests) and fix them

> 📌 **Illustrative scenario: Lea, junior developer at a SaaS startup**
> Lea asks "refactor the code" and gets inconsistent changes across 12 files. After reading WP #1, she reformulates: "Extract validation logic from `OrderService.ts` into a `validators/order.ts` module, keep existing tests green." Result: a targeted diff, accepted on the first try.

4.2 #2: Customizing Claude (~20 min)

→ See **#2 Customizing Claude** (`02-customization.qmd`)

Audience: Developers looking to optimize their setup.

- Create an effective project `CLAUDE.md` with rules, tech stack, and code conventions
- Configure reusable specialized Agents (test, review, documentation)
- Connect Skills for persistent business knowledge across sessions

Customization is the lever that transforms Claude Code from a generic tool into a teammate that knows your conventions. A well-crafted `CLAUDE.md` eliminates repetitive reminders and reduces noise in every interaction.

4.3 #3: Security in Production (~25 min)

→ See **#3 Security in Production** (03-security.qmd)

Audience: DevSecOps, Tech Leads, teams in sensitive environments.

- Configure `permissions.deny` to block dangerous commands in production
- Implement pre/post-execution validation hooks for auditability
- Protect secrets and API keys from inadvertent leakage into context

> 📌 **Scenario: Karim, DevSecOps at a fintech (50 devs)**
> Karim discovers that a developer inadvertently loaded a `.env` containing Stripe keys into Claude's context. After reading WP #3, he deploys a `pre-tool-use` hook that automatically blocks reading any file matching `*.env*` or `*secret*`. Result: keys no longer leave the workstation, and the audit team maintains compliance evidence.

4.4 #4: Architecture Demystified (~20 min)

→ See **#4 Architecture Demystified** (04-architecture.qmd)

Audience: Architects, developers curious about the internals.

- Understand the agent loop (perception → planning → execution → feedback)
- Know why Claude "forgets" between sessions and how to compensate with CLAUDE.md
- Identify system limits to calibrate your expectations and prompts

Understanding the internal architecture is not an academic exercise: knowing that Claude operates in an agentic loop (rather than simple request-response) changes how you structure your requests and manage long-running tasks.

4.5 #5: Deploying Across Teams (~20 min)

→ See **#5 Deploying Across Teams** (05-team.qmd)

Audience: Tech Leads, Managers, growing teams.

- Set up a shared CLAUDE.md versioned in Git to align the entire team
- Connect Claude Code to your CI/CD pipeline for automated reviews and checks
- Measure adoption and impact with available observability tools

> 📌 **Scenario: Sophie, Tech Lead at an e-commerce SMB (12 devs)**
> Sophie notices each developer uses Claude Code differently: inconsistent naming conventions, tests sometimes skipped. After WP #5, she sets up a shared `CLAUDE.md` in the

monorepo with team conventions, and a `reviewer` agent in the CI pipeline. Result: Claude-generated PRs meet standards on the first submission.

4.6 #6: Privacy & Compliance (~15 min)

→ See **#6 Privacy & Compliance** (`06-privacy.qmd`)

Audience: Compliance officers, enterprise teams.

- List exactly which data leaves for Anthropic during a session
- Configure available retention rules and opt-outs for your organization
- Assess GDPR compliance and hosting options based on your sector

This is often the first question asked by management before any adoption: "what data leaves our servers?". This whitepaper provides factual answers to unblock internal validations.

4.7 #7: The Reference Guide (~2h)

→ See **#7 The Reference Guide** (`07-guide-reference.qmd`)

Audience: All profiles, consolidated reference document.

- Find any command, configuration, or workflow in seconds
- Access the 5 exclusive chapters (Migration, IDE, Git, Patterns, Methodologies)
- Use as a daily reference: complete index with line numbers

📌 Memo: Choose your path in 30 seconds

Solo dev getting started: #0 then #1 is enough to be productive. Add #2 when you want to customize.

Team lead: Start with #5 (team deployment) then #3 (security). #2 will come naturally when the team standardizes its workflows.

CTO / Decision-maker: #6 (privacy/compliance) first, then #5 for rollout. #4 (architecture) helps calibrate technical expectations.

5 Recommended paths

There is no single right reading order: the best path depends on your role, your experience level with Claude Code, and your most pressing constraint (security, team adoption, compliance). The decision tree below provides a quick orientation, followed by a summary table organized by profile.

```
I want to use Claude Code
└─ My context?
   ├─ Urgent security / exposed secrets ⟶ #3 Security
   ├─ Discovering the tool ⟶ My role?
   │  ├─ Junior developer ⟶ #1 Prompts ⟶ #2
   │  │  └─ #7 Reference if consolidation needed
   │  ├─ Senior developer ⟶ #1 ⟶ #2 ⟶ #3 ⟶ #4
   │  │  └─ #7 Reference if consolidation needed
   │  ├─ Architect / curious ⟶ #4 Architecture ⟶ #3
   │  │  └─ #7 Reference if consolidation needed
   │  └─ CTO / decision-maker ⟶ #6 ⟶ #5 ⟶ #4
   ├─ I have a team to align ⟶ #5 Team ⟶ #3 ⟶ #2
   │  └─ #7 Reference if consolidation needed
   └─ GDPR / compliance question ⟶ #6 Privacy ⟶ #5
```

Profile	Path	Time
Junior	#0 → #1	~25 min
Senior	#0 → #1 → #2 → #3	~1h15
Power User	#0 → #4 → #3 → #5	~1h15
Tech Lead	#0 → #5 → #3 → #2	~1h15
PM/Manager	#0 → #5 → #6	~45 min
Compliance	#0 → #6 → #3	~50 min

Alternative: The **#7 Reference Guide** (~2h) consolidates all whitepapers + 5 exclusive chapters (Migration, IDE, Git, Patterns, Methodologies).

The **#8 Agent Teams** covers experimental multi-agent orchestration (v2.1.32+).

5.1 Solo vs team vs enterprise

Configuration and governance needs scale with team size. A solo developer can get by with a project-level `CLAUDE.md`, while a team of 20+ will need centralized permissions, security hooks,

and compliance rules. This table summarizes the key differences to help you gauge your initial investment.

Dimension	Solo dev	Team (2-20)	Enterprise (20+)
Key whitepaper	#1, #2	#5, #3	#6, #5, #3
Minimum config	Project `CLAUDE.md`	Shared `CLAUDE.md` in Git	`CLAUDE.md` + centralized permissions
Security	Best practices #3	CI hooks + deny list	Audit trail + GDPR compliance
CI/CD	Optional	Recommended (reviewer agent)	Mandatory
Setup time	30 min	2-4h	1-2 days

6 Resources

Repository: github.com/FlorianBruniaux/claude-code-ultimate-guide

- Full guide (26,648 lines)
- Printable cheatsheet
- 271 templates (agents, skills, commands, hooks, rules)
- Advanced workflows and methodologies

Official documentation: docs.anthropic.com/en/docs/claude-code

6.1 Reliability conventions

Not all information in this series carries the same weight. Official Anthropic documentation is treated as the source of truth; observations from independent testing or community feedback are explicitly tagged with their confidence level.

Tier	Source	Reliability
Tier 1	Anthropic documentation	Official claim, version and provider scope to verify
Tier 2	Verified tests	Observation limited to the tested conditions
Tier 3	Community inference	Hypothesis requiring local validation

Tier 2/3 information is explicitly marked.

Type	License
Content	CC BY-SA 4.0
Code	MIT

Version 3.44.1 | October 2026

7 The series

This whitepaper is part of the **Claude Code Ultimate Guide** series, a complete resource for mastering Claude Code, from installation to production deployment.

13 whitepapers in the series:

- **WP00** Introduction & Fundamentals
- **WP01** Effective Prompts
- **WP02** Customization & Configuration
- **WP03** Security in Production
- **WP04** Internal Architecture
- **WP05** Team Adoption
- **WP06** Privacy & Compliance
- **WP07** Condensed Reference Guide
- **WP08** Agent Teams & Sub-Agents
- **WP09** Learning with AI (UVAL)
- **WP10** AI Budget & ROI
- **WP11** Team Metrics in the AI Era
- **WP12** Software Engineering in the Age of Autonomous Agents

Full series available at cc.bruniaux.com/whitepapers/