Gemini CLI: Configuration & GEMINI.md Guide

Bootstrap Your Project with Free AI Assistance

Out The Mud Work Shops

🎯 Why Gemini Configuration Matters

Your mud isn't just writing code - it's explaining your project's context every single prompt. With proper Gemini CLI configuration, you explain once and Gemini remembers throughout your session.

🚀 Quick Start: Bootstrap with `/init`

The Magic `/init` Command

Start Gemini CLI in your project root and run:

```
$ gemini
> /init
```

What `/init` Does

```
> /init

Gemini: Analyzing project structure...
✓ Detected: Node.js + Express + MongoDB
✓ Found test framework: Jest
✓ Identified patterns: MVC architecture
✓ Located config files: .env, .eslintrc
✓ Analyzed code style: ES6+, async/await

Creating initial configuration:
✓ Generated GEMINI.md with project overview
✓ Created .geminiignore based on .gitignore
✓ Detected coding conventions
✓ Set up initial context

Your project is ready! I understand:
- Architecture: MVC pattern
- Database: MongoDB with Mongoose
- Testing: Jest with supertest
- Style: Airbnb ESLint rules
```

Auto-Generated GEMINI.md

```
# Project: [From package.json]

## Technology Stack
- Runtime: Node.js v20.x
- Framework: Express 4.x
- Database: MongoDB with Mongoose
- Testing: Jest + Supertest
```

```
- Linting: ESLint (Airbnb config)

## Project Structure
- `/src` - Source code
  - `/models` - Mongoose schemas
  - `/routes` - Express routes
  - `/controllers` - Route handlers
  - `/middleware` - Custom middleware
  - `/utils` - Helper functions
- `/tests` - Test files
- `/config` - Configuration files

## Detected Patterns
- Async/await for asynchronous code
- JWT authentication
- Error handling middleware
- RESTful API design

## Commands
- `npm start` - Start server
- `npm test` - Run tests
- `npm run dev` - Development mode
- `npm run lint` - Run ESLint
```

📋 The GEMINI.md File

What is GEMINI.md?

Your project's AI instruction manual. Place it in your project root to give Gemini persistent context about your codebase, conventions, and requirements.

Basic GEMINI.md Template

```
# Project: E-Commerce API

## Overview
Multi-tenant e-commerce platform with real-time inventory tracking.

## Critical Rules
1. All monetary values in cents (integers only)
2. Use database transactions for orders
3. Soft delete only (deleted_at timestamp)
4. API responses must include request_id

## Code Style
- Functional programming preferred
- No var declarations (const/let only)
- Async/await over promises
- 2-space indentation

## Architecture
- Controllers: Thin, routing only
- Services: Business logic
- Repositories: Database access
- Validators: Input validation
```

```
## Testing Requirements
- Unit tests for all services
- Integration tests for all endpoints
- Minimum 80% coverage
- Use factories for test data

## Security
- Validate all inputs
- Use parameterized queries
- Rate limit all endpoints
- Log security events
```

🔧 Advanced GEMINI.md Examples

For Full-Stack Projects

```
# Full-Stack SaaS Application

## Frontend (React)
- Components: Functional with hooks
- State: Redux Toolkit
- Styling: Tailwind CSS
- Testing: React Testing Library

## Backend (Node.js)
- Framework: NestJS
- Database: PostgreSQL
- ORM: Prisma
- Auth: JWT + refresh tokens

## Shared Conventions
- TypeScript everywhere
- Monorepo with npm workspaces
- Shared types in packages/types
- API contracts in OpenAPI 3.0

## Development Workflow
1. Create feature branch
2. Update API spec first
3. Generate types from spec
4. TDD for new features
5. PR requires 2 approvals

## Common Tasks
When adding a new feature:
1. Start with database schema
2. Update Prisma schema
3. Generate migrations
4. Create service with tests
5. Add controller endpoints
6. Update API documentation
7. Implement frontend
```

For Microservices

```
# Microservices Architecture

## Service Boundaries
- Auth Service: Port 3001
- User Service: Port 3002
- Order Service: Port 3003
- Payment Service: Port 3004

## Communication
- Sync: REST with circuit breakers
- Async: RabbitMQ for events
- Service discovery: Consul

## Shared Libraries
- @company/logger
- @company/auth-middleware
- @company/error-handler

## Database Strategy
- One database per service
- No shared databases
- Event sourcing for order service
- Read replicas for reporting

## Deployment
- Docker containers
- Kubernetes orchestration
- GitHub Actions CI/CD
- Blue-green deployments
```

📁 Configuration Hierarchy

1. Global Configuration

`~/.gemini/GEMINI.md` - Rules for all your projects

```
# Global Development Preferences

## General Principles
- Prefer composition over inheritance
- Write tests before implementation
- Document public APIs
- Use semantic versioning

## Preferred Tools
- Package manager: npm (not yarn)
- Testing: Jest
- Linting: ESLint + Prettier
- Git: Conventional commits
```

2. Project Configuration

`./GEMINI.md` - Project-specific rules

```
# Project: Customer Portal

## Overrides Global
- Package manager: pnpm (exception)
- Testing: Vitest (replacing Jest)

[Project specific rules...]
```

3. Module Configuration

`./src/modules/auth/GEMINI.md` - Module-specific context

```
# Auth Module

## Special Considerations
- High security requirements
- 2FA implementation required
- Session timeout: 15 minutes
- Password requirements: NIST guidelines

## Key Files
- `auth.service.ts` - Core authentication
- `token.service.ts` - JWT management
- `2fa.service.ts` - Two-factor auth
```

🛠️ settings.json Configuration

Location

- User: `~/.gemini/settings.json`
- Project: `./.gemini/settings.json`

Basic settings.json

```
{
  "model": "gemini-2-flash",
  "temperature": 0.7,
  "contextFileName": "GEMINI.md",
  "confirmBeforeRunning": true,
  "excludeTools": ["run_shell_command(rm -rf)"],
  "theme": "dark",
  "mcpServers": {}
}
```

Advanced Configuration

```
{
  "model": "gemini-2-flash",
  "temperature": 0.5,
  "safetySettings": [{
    "category": "HARM_CATEGORY_DANGEROUS_CONTENT",
```

```
    "threshold": "BLOCK_ONLY_HIGH"
  }],
  "coreTools": [
    "list_directory",
    "read_file",
    "write_file",
    "search_file_content"
  ],
  "excludeTools": [
    "run_shell_command(rm)",
    "run_shell_command(sudo)"
  ],
  "mcpServers": {
    "github": {
      "command": "npx",
      "args": ["@modelcontextprotocol/server-github"],
      "env": {
        "GITHUB_TOKEN": "${GITHUB_TOKEN}"
      }
    }
  },
  "autoApprove": {
    "read": true,
    "write": false,
    "shell": false
  }
}
```

🚫 The .geminiignore File

Purpose

Like `.gitignore`, but for Gemini CLI. Prevents access to sensitive or irrelevant files.

Example .geminiignore

```
# Dependencies
node_modules/
vendor/
*.lock

# Build outputs
dist/
build/
coverage/

# Sensitive files
.env
.env.*
*.pem
*.key

# Large files
*.log
*.sql
```

```
*.dump

# IDE
.vscode/
.idea/

# OS
.DS_Store
Thumbs.db
```

🔄 Common Configuration Scenarios

Scenario 1: Onboarding New Developer

```
> /init

Gemini: I've analyzed your codebase. Would you like me to:
1. Explain the architecture
2. Set up your development environment
3. Show common workflows
4. All of the above

> "All of the above"
```

Scenario 2: Legacy Code Migration

```
# GEMINI.md
## Migration In Progress
- Old code: /src/legacy (ES5, callbacks)
- New code: /src/v2 (ES6+, async/await)
- Migration guide: /docs/migration.md

## When working on legacy code:
1. Don't refactor without tests
2. Document current behavior first
3. Migrate in small chunks
4. Keep both versions working
```

Scenario 3: Multi-Language Project

```
# GEMINI.md
## Language Conventions
- Python: /backend (PEP 8, type hints)
- TypeScript: /frontend (strict mode)
- Go: /services (standard formatting)

## Cross-language rules:
- API contracts in OpenAPI
- Shared types generated from spec
- Consistent error codes
```

📊 Configuration Best Practices

Do's ✅

- Run `/init` on existing projects
- Keep GEMINI.md under version control
- Update configuration as project evolves
- Use module-specific configs for complex areas
- Document "why" not just "what"

Don'ts ❌

- Don't include secrets in GEMINI.md
- Don't make configs too verbose
- Don't forget to update after major changes
- Don't ignore .geminiignore for sensitive files
- Don't duplicate obvious information

🎯 This Week's Challenge

Configuration Mastery Challenge:

1. Run `/init` on your main project
2. Enhance the generated GEMINI.md
3. Add module-specific configurations
4. Create helpful .geminiignore
5. Test with complex requests

Measure the difference in response quality!

💡 Pro Tips

Dynamic Context Loading

```
# Check what context is loaded
> /memory show

# Refresh after changes
> /memory refresh
```

Testing Your Configuration

```
> "Explain our coding standards"
# Should reflect your GEMINI.md content

> "What's our testing strategy?"
# Should match your documented approach
```

Team Alignment

```
# GEMINI.md
## For New Team Members
When you join, Gemini can:
```

```
- Explain our architecture: "Explain the system design"
- Show coding standards: "Show me example code"
- Demo workflows: "Walk me through adding a feature"
```

Remember: A well-configured Gemini CLI is like having a senior developer who perfectly knows your codebase and never forgets your standards. Configure once, accelerate forever.

Rise above repetitive context. Let configuration be your ladder out of the mud.