

한 걸음 앞선 개발자가 지금 꼭 알아야 할 **클로드 코드**

한 걸음 앞선 개발자가

지금 꼭 알아야 할

MASTERING
CLAUDE
CODE

조훈, 정찬훈 지음

클로드

코드

CLAUDE
CODE

실무에서 검증된 개발 방식 그대로,

매일 1시간 4주

Claude Code 에이전트 실전 훈련!

```
# 설치(macOS, Windows)
# CLAUDE.md 설정
# MCP 연동/다양한 활용 전략
# 클로드 워크플로 전략
# 설계 → 부트스트래핑 →
  테스트 → 개선 → 명세
# 클로드 코드의 효율을 극대화
  하는 다양한 방법
```

갈벗

깃허브: https://github.com/sysnet4admin/_Book_Claude-Code

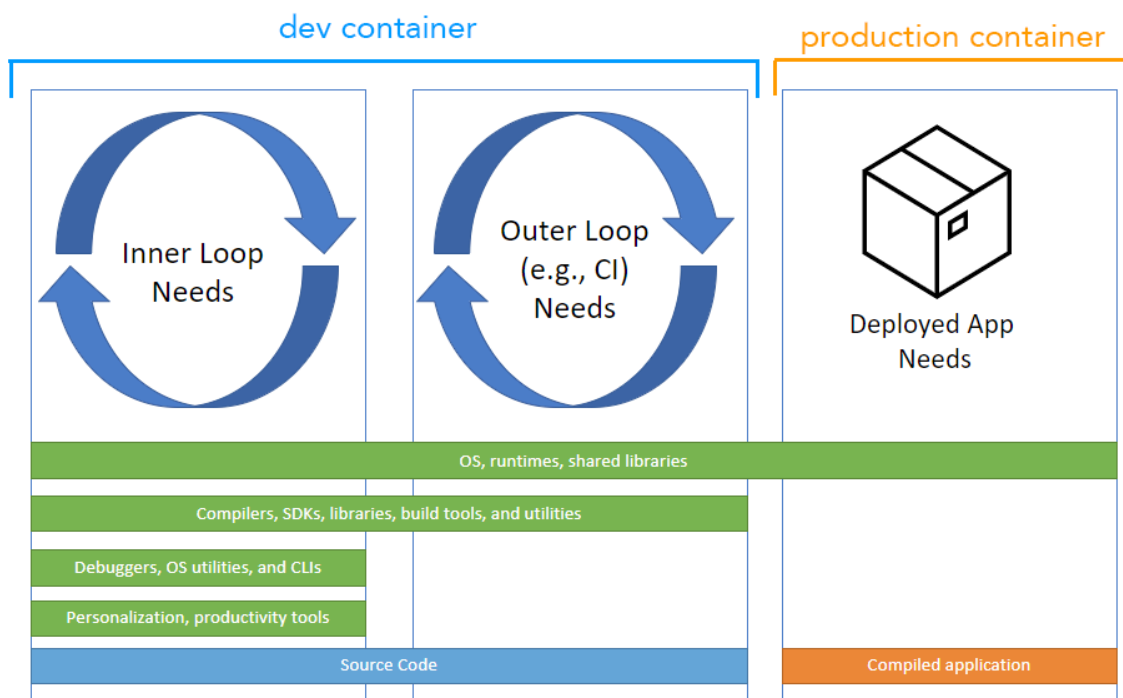
책에서는 Dev Container로 설명을 시작하지만 이 문서에서는 Dev Container를 생성하는 도구 중에 하나인 DevPod로 시작하도록 하겠습니다. Dev Container에 대해서는 책에서 이미 설명되었으며, 이 문서의 주요 목적은 Dev Container와 DevPod를 통해 독립된 클로드 코드의 실습 환경을 구성하는 것이 목적이기 때문입니다.

실습을 진행하기에 앞서 책에서는 지면의 한계로 설명하지 못했던 Dev Container가 나오게 된 배경과 Dev Container를 위해 DevPod를 선택한 이유를 우선 설명하도록 하겠습니다.

1. Dev Container를 위한 DevPod

프로젝트를 진행하다보면, 프로젝트마다 고유의 환경이 필요하고 이를 각 개발자가 문서만 보고 구성 관리하는 것에는 어려움이 있습니다. 또한 개발(dev) 환경에서 작성한 내용이 문제 없이 운영(production)으로 넘어가기 위한 과정도 순조롭지 않을 수 있을 수 있습니다. 따라서 이를 표준화하고 모두가 동일한 환경에서 코드를 작성하는 것이 매우 중요한 부분입니다.

[그림 1] 프로젝트 진행에 따른 dev container 그리고 production container로 넘어가는 과정



이러한 요구 사항을 위해 초기에는 제공되는 문서 또는 스크립트 등을 사용해 환경을 만들었으며, 이후 개발을 위한 표준화된 가상 머신을 제공하였습니다. 그 다음으로 더 가벼워지고 플랫폼에 종속성을 벗어난 컨테이너로 개발 환경을 제공하게 되었습니다.

즉 프로젝트의 통일된 진행을 위해 **개발 환경 표준화**를 하려고 했던 과정을 정리하면 다음과 같습니다.

1세대: 셸 스크립트 또는 Git(or SVN)과 같은 소스 관리 도구로 개발 환경의 일관성을 유지

2세대: 가상화 기술을 사용해 관리자가 표준 개발 환경이 구성된 가상 머신을 개발자에 제공

3세대: 컨테이너 기술을 사용해 개발자가 직접 표준화된 개발 환경을 구성할 수 있도록 함

이러한 활동을 모두 통칭하여 코드형 개발 환경(DevEnv as Code)이라고 부릅니다.

동시에 베이그런트를 시작으로 테라폼과 같은 코드형 인프라(IaC, Infrastructure as Code) 도구가 보편화 되며, 개발 환경도 표준화된 인프라를 코드로 보급하고자 하는 시도가 진행되었습니다. 특히 개발 환경 표준화를 코드로 진행하는 것이 더 효율적이었던 이유는 각 프로젝트마다 요구하는 라이브러리, 프레임워크, 그 외에 다양한 환경 조건을 가장 잘 이해하고 있는 사람은 개발자였고, 이러한 내용을 가장 손쉽게 작성할 수 있는 것이 코드였기 때문입니다. 이러한 요구 사항과 3세대 기술이 결합되며, 코드형 개발 환경(DevEnv as Code)이 개발 (환경을 가진) 컨테이너 ([Development Container](#))로 발전하게 되었습니다.

개발 컨테이너에 대해서 추가 설명을 하면 그 필요성에 대해서 충분한 이해가 되실 것 같습니다. 3세대 내에서 사용된 도커(Docker)가 제공하는 것은 단순히 컨테이너 런타임을 이용해 인프라적인 요건들만을 충족하는 것이기 때문에 개발자의 다양한 요구 사항이 담긴 “개발 환경 표준화”를 만족시킬 수 없었습니다.

예를 들면, 다음과 같은 요구 사항이 도커에 비해 더 있을 수 있습니다.

1. 통합 개발 도구 (IDE, Integrated Development Environment) 지원
2. 쿠버네티스 및 클라우드 리소스 사용
3. 팀/조직 간의 협업 기능
4. 보안 기능

해당 기능들은 도커만으로는 구현하기 어려워 이를 더 고도화하기 위한 여러 노력이 있었고, 그러한 산물로 개발 컨테이너라고 하는 표준화가 진행된 것입니다. 이를 기반으로 다양한 형태의 개발 환경 표준화 도구가 나오게 되었고, 각 도구마다 장점과 단점이 존재합니다.

우선 대표적인 도구 3개를 표를 통해서 비교해 보도록 하겠습니다.

구분	VS Code Dev Containers Extension	Dev Container CLI	DevPod
주요 목적	VS Code에서 Devcontainer 기반 원격 개발 환경 제공	🔧 CLI로 Dev Container 기반 이미지 빌드 및 실행	🌐 다양한 환경에서 Dev Container 구동 및 자동화된 통합 지원
IDE 지원	🚫 VS Code 전용	🚫 CLI 컨테이너만 띄움	🎯 IDE 독립, VS Code, JetBrains, Vim 등 다양한 통합 개발 도구 지원
개발환경 인프라 지원	🐳 Docker 호환 컨테이너 런타임	🐳 Docker 호환 컨테이너 런타임	☁️ Docker, Kubernetes 원격, SSH 서버, 커스텀 프로바이더 작성 등 다양한 지원
주요 장점	🔗 VS Code와의 연동, 손쉬운 개발자 경험	⚡ IDE 독립성, 가벼움, CI/CD 자동화가 쉬움	🖥️ 다양한 인프라 및 IDE 지원, GUI 및 CLI 기반 통합 관리 인터페이스 제공
적합한 환경	👤 VS Code를 사용하여 단독 개발 하는 경우	🚚 CI/CD 파이프라인과 개발 환경을 빠르게 전환 적용해야 하는 경우	🏢 팀/조직 단위 개발환경을 관리해야 하는 경우
한계점	📍 VS Code에 종속	⚠️ Docker 호환 컨테이너 외 확장성에 편의 기능 부족	📦 제공자를 선택하고 관리해야 함

각 도구는 모두 Dev Container를 위한 지원은 충분히 제공되나, DevPod의 경우 붉은 글씨로 표시한 것처럼 다양한 인프라 환경 지원 및 IDE 통합이 가능하다는 점에서 좀 더 효과적인 실습이 가능합니다.

물론 VS Code 또는 Cursor에서는 손쉽게 Dev Container가 구동할 수 있습니다. 하지만 도커 외에 다른 컨테이너 런타임이 지원되지 않고, 각 IDE에 제한된다는 점에서 DevPod를 먼저 학습하고 필요에 따라

선택하는 것이 가장 좋을 것 같습니다. Dev Container CLI의 경우에는 사용 목적이 다소 제한적이므로 현재 실습에는 적합하지 않으며, DevPod에 익숙해지고 나면 필요할 때 사용하는 것이 어렵지 않으실 것입니다.

지금까지 Dev Container 그리고 그것을 구현한 DevPod를 선택해서 사용한다고 했는데 클로드 코드 관점에서 왜 Dev Container를 사용하는지를 설명하고 실습을 진행하겠습니다.

클로드 코드는 이미 지금까지 충분히 실습한 것과 같이 데스크탑/랩탑에 있는 모든 정보를 읽을 수 있고, 해당 내용을 학습하여 작업을 진행할 수 있습니다. 이는 매우 강력하고 그만큼 유용하나, 클로드 코드가 실행하는 내용을 사용자가 유심히 보지 않고 진행시키거나, YOLO(--dangerously-skip-permissions) 모드를 사용하여 진행하는 경우 학습하고 진행하지 않아야 하는 내용들이 포함될 수 있습니다.

이와 같은 경우를 방지하고자 CLAUDE.md와 설정 값들을 조정할 수 있지만, 가장 강력한 방법은 호스트 시스템과 분리하는 것입니다. 그러한 목적으로 앤트로픽에서도 클로드 코드 사용 시에 독립적인 환경을 구축하는 것을 권고하며, 그에 따라 Dev Container를 구축하는 실습을 진행한다고 이해하시면 될 것 같습니다. 참고로 클로드 코드는 **VS Code Dev Containers Extension** 을 권장하지만, 다양한 실습 환경 체험을 위해 **DevPod**로 진행한다는 점을 다시 한번 말씀드립니다.

그러면 충분히 왜 Dev Container를 사용하기 위해 DevPod를 사용하는지를 설명하였으니 이제 실습을 진행해 보도록 하겠습니다.

2. DevPod 설치와 기본 구성

DevPod 설치와 구성은 매우 간단합니다. 이 문서에는 macOS를 기준으로 진행하며, 우분투의 환경에서는 추가적인 설정이 필요할 수 있음을 미리 안내 드립니다.

1. **brew install --cask devpod** 를 통해서 DevPod 도구를 설치합니다.

<Terminal박스>

```
$ brew install --cask devpod

==> Auto-updating Homebrew...

[중략]

==> Downloading
https://github.com/loft-sh/devpod/releases/download/v0.6.15/DevPod_macos_aarch64.dmg

==> Downloading from
https://release-assets.githubusercontent.com/github-production-release-asset/592748160/f176

#####
##### 100.0%
```

```
==> Installing Cask devpod
```

```
==> Moving App 'DevPod.app' to '/Applications/DevPod.app'
```

```
==> Linking Binary 'devpod-cli' to '/opt/homebrew/bin/devpod'
```

[생략]

</Terminal박스>

2. 설치 후에 동작 확인을 위해 **devpod --help**를 입력합니다. 이 중에 실습에서 사용할 명령어는 delete, ide, list, provider, ssh, up 정도 입니다. 따라서 별로 복잡하지 않습니다.

<Terminal박스>

```
$ devpod --help
```

```
DevPod
```

```
Usage:
```

```
devpod [command]
```

```
Available Commands:
```

```
build      Builds a workspace
```

```
completion Generate the autocompletion script for the specified shell
```

```
context    DevPod Context commands
```

```
delete     Deletes an existing workspace
```

```
help       Help about any command
```

```
ide        DevPod IDE commands
```

```
list       Lists existing workspaces
```

```
logs       Prints the workspace logs on the machine
```

```
logs-daemon Prints the daemon logs on the machine
```

machine	DevPod Machine commands
pro	DevPod Pro commands
provider	DevPod Provider commands
ssh	Starts a new ssh session to a workspace
status	Shows the status of a workspace
stop	Stops an existing workspace
up	Starts a new workspace
upgrade	Upgrade the DevPod CLI to the newest version
use	Use DevPod resources
version	Prints the version

</Terminal박스>

3. devcontainer를 생성하기 위해 직접 관련 파일들을 만들어야 하지만, 이는 편의적인 목적으로 2주차 수요일의 devcontainer에 관련 파일들을 미리 만들어 두었습니다. 따라서 2주차 수요일 devcontainer 디렉터리로 이동하고 **ls -l .devcontainer** 명령을 통해서 해당 내용이 다음과 같은지 확인합니다. 파일 내용에서 확인할 수 있는 것처럼 도커는 미리 설치 구동되어 있어야 합니다.

<Terminal박스>

```
$ cd week2/Wed/devcontainer

$ ls -l .devcontainer

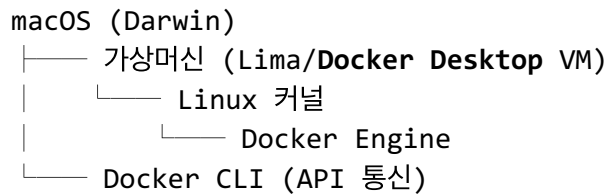
Dockerfile          # Node.js 20 기반의 개발 컨테이너 이미지 빌드 설정 파일
devcontainer-cursor.json # Cursor 에디터용 Claude Code Sandbox 개발 컨테이너 설정 파일
devcontainer-on-k8s.json # k8s 환경에서 실행되는 Claude Code Sandbox 개발 컨테이너 설정 파일
devcontainer.json    # 기본 Claude Code Sandbox 개발 컨테이너 설정 파일
init-firewall.sh     # Docker DNS 정보 추출 및 방화벽 초기화를 위한 bash 스크립트
```

</Terminal박스>

<노트> 회사 맥북에서 라이선스 이슈로 **docker** 실행을 위한 도커 데스크탑을 설치할 수 없어요

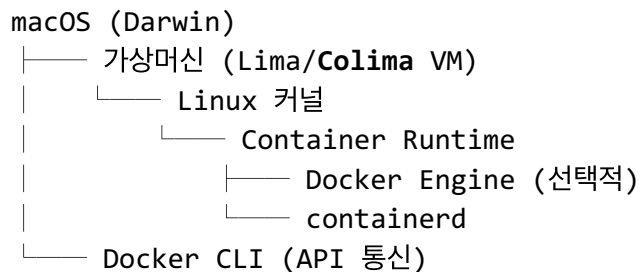
일반적으로 macOS에서는 커널의 차이로 **docker** 명령을 직접 실행할 수 없습니다. **docker**는 리눅스의 커널 기능은 **cgroups**, **namespaces** 등을 사용해서 구현되었기 때문입니다.

따라서 이런 경우 개인용 랩탑에서는 도커 데스크탑을 이용해서 간단히 해결할 수 있으나,



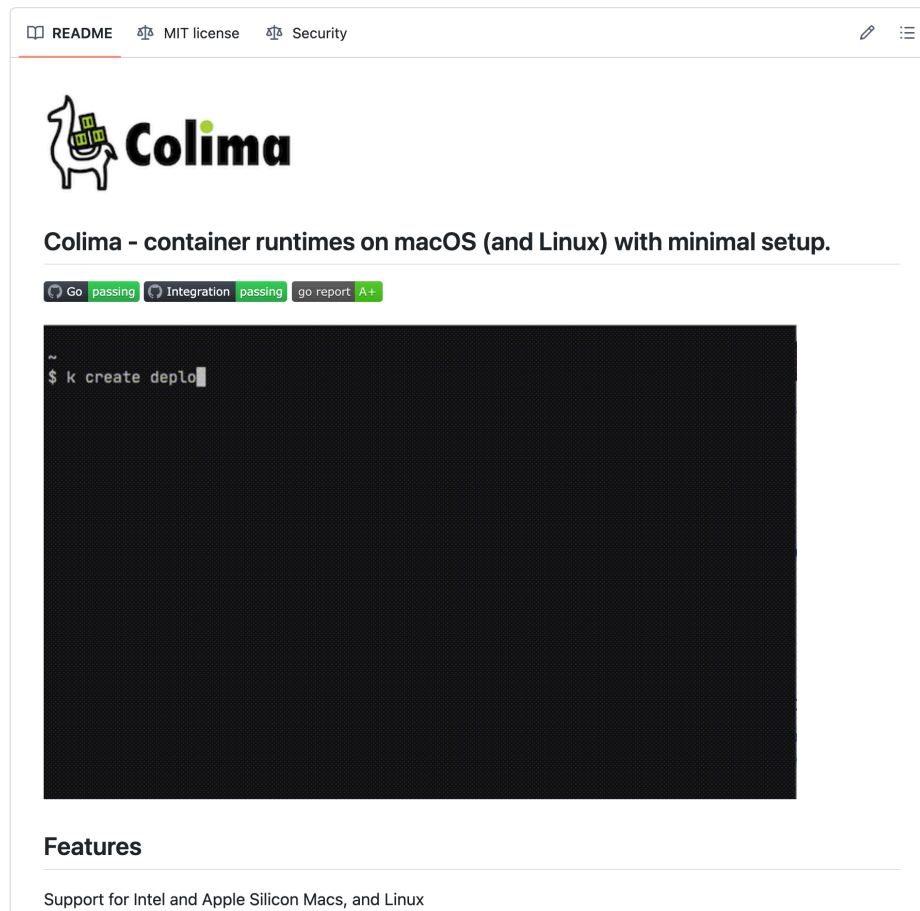
도커 데스크탑은 라이선스 제한 조건이 있기 때문에 회사의 환경에서는 사용하기 어렵습니다.

이런 경우 다양한 오픈 소스 프로젝트들을 이용해서 도커 데스크탑과 유사한 환경을 구성할 수 있습니다. 그 중 대표적인 오픈 소스 프로젝트는 **Colima**로 현재 24.8k의 별을 가지고 있을 정도로 인기 있는 프로젝트이며, 사례도 굉장히 많은 도구이므로 이를 통해 macOS에서 **docker**를 실행하시기 바랍니다.



참고로 **회사마다 정책은 매우 다르므로** 오픈 소스라고 해도 **colima**와 **docker** 런타임을 사용할 수 없는 경우가 있으니 꼭 해당 부분은 따로 확인하셔야 합니다.

[그림 2] Colima의 README.md (<https://github.com/abiosoft/colima>)



기본 설치는 매우 간단하며, docker 런타임을 그대로 사용하므로 이것도 함께 설치되어야 합니다.

<Terminal박스>

```
$ brew install docker
```

[생략]

```
$ brew install colima
```

[생략]

</Terminal박스>

다만 이 경우 환경에 따라 colima를 통한 docker 호출에 추가 구성이 필요할 수 있습니다. 이것은 클로드 코드에 맡겨 편리하게 구성하시기 바랍니다.

</노트>

4. DevPod가 devcontainer를 구성하기 위해서는 어떤 제공자를 통해서 컨테이너를 구성할지 정해주어야 합니다. 따라서 **devpod provider add docker**로 docker를 제공자로 설정하고, **devpod provider use docker**로 docker를 기본 제공자로 인식하도록 합니다.

<Terminal박스>

```
$ devpod provider add docker
```

```
15:45:25 done Successfully installed provider docker
```

```
15:45:25 done Successfully configured provider 'docker'
```

```
$ devpod provider use docker
```

```
15:45:35 info To reconfigure provider docker, run with '--reconfigure' to reconfigure the provider
```

```
15:45:35 done Successfully switched default provider to 'docker'
```

</Terminal박스>

5. 자 이제 DevPod를 통해 준비된 파일들을 호출하여 devcontainer를 구성하는 단계만 남았습니다. **devpod up .** 을 실행하면, 자동으로 현재 숨겨진 .devcontainer 디렉터리에 있는 파일들을 읽어 devcontainer를 구성합니다. 만약 VS Code가 사전에 설치되어 있다면 VS Code 터미널에 자동 연결되는 것까지 확인할 수 있을 것입니다. 호스트 환경에 따라 VS Code의 터미널은 한번에 자동 연결되지 않을 수 있으나 리모트 터미널(주소: devcontainer.devpod) 연결을 다시 시도해 보시기 바랍니다.

<Terminal박스>

```
$ devpod up .
```

```
16:12:47 info Creating devcontainer...
```

```
16:12:50 info Build with docker buildx...
```

```
16:12:51 info #0 building with "container" instance using docker-container driver
```

```
16:12:51 info
```

```
16:12:51 info #1 [internal] load build definition from Dockerfile
```

```
16:12:51 info #1 transferring dockerfile: 2.59kB done
```

```
16:12:51 info #1 DONE 0.0s
```

```
[중략]
```

```
16:13:52 info Adding 13.107.213.74 for update.code.visualstudio.com
```

```
16:13:52 info Host network detected as: 172.17.0.0/24

16:13:52 info Firewall configuration complete

16:13:52 info Verifying firewall rules...

16:13:55 info Firewall verification passed - unable to reach https://example.com as expected

16:13:55 info Firewall verification passed - able to reach https://api.github.com as expected

16:13:55 done Successfully ran command : sudo /usr/local/bin/init-firewall.sh

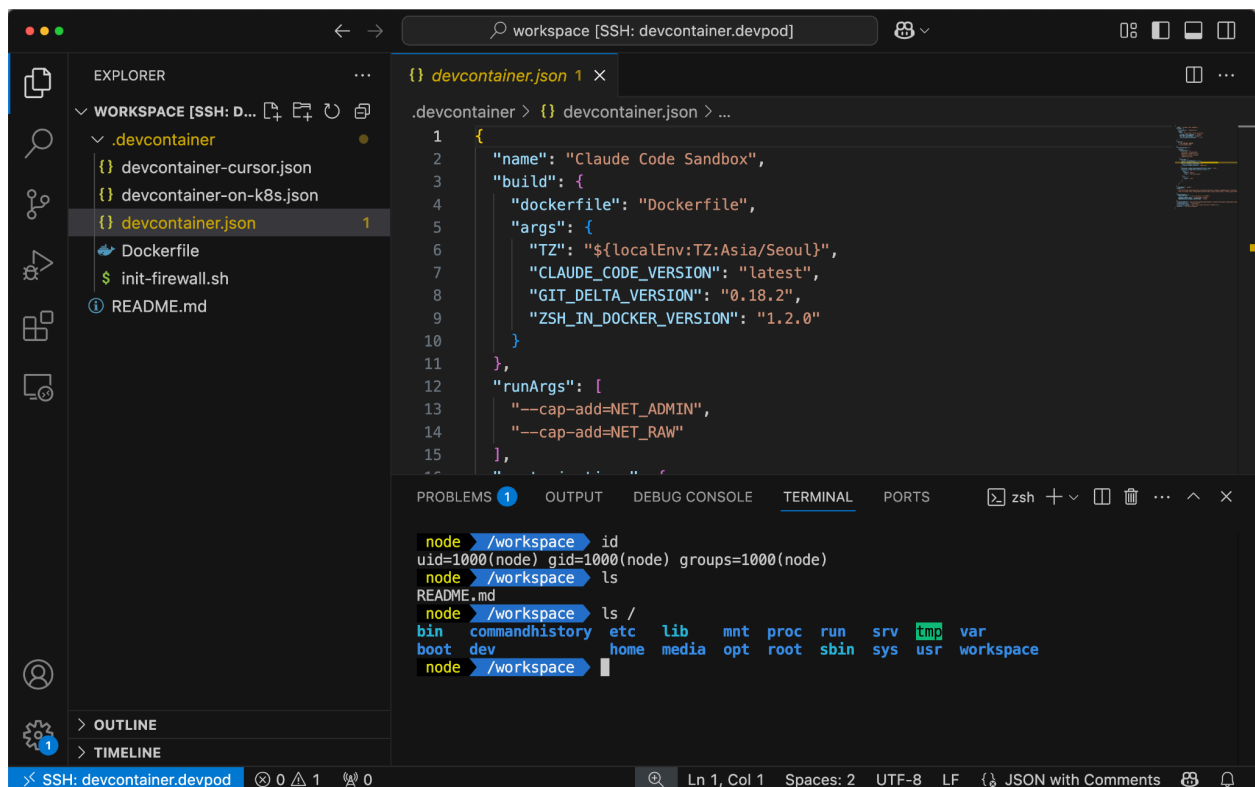
16:13:55 info Install extensions
'anthropic.claude-code,dbaeumer.vscode-eslint,esbenp.prettier-vscode,eamodio.gitlens' in the
background

16:13:55 info Run 'ssh devcontainer.devpod' to ssh into the devcontainer

16:13:55 info Starting VSCode...
```

</Terminal박스>

[그림 3] devpod로 생성된 devcontainer가 자동으로 VS Code 터미널에 연결됨



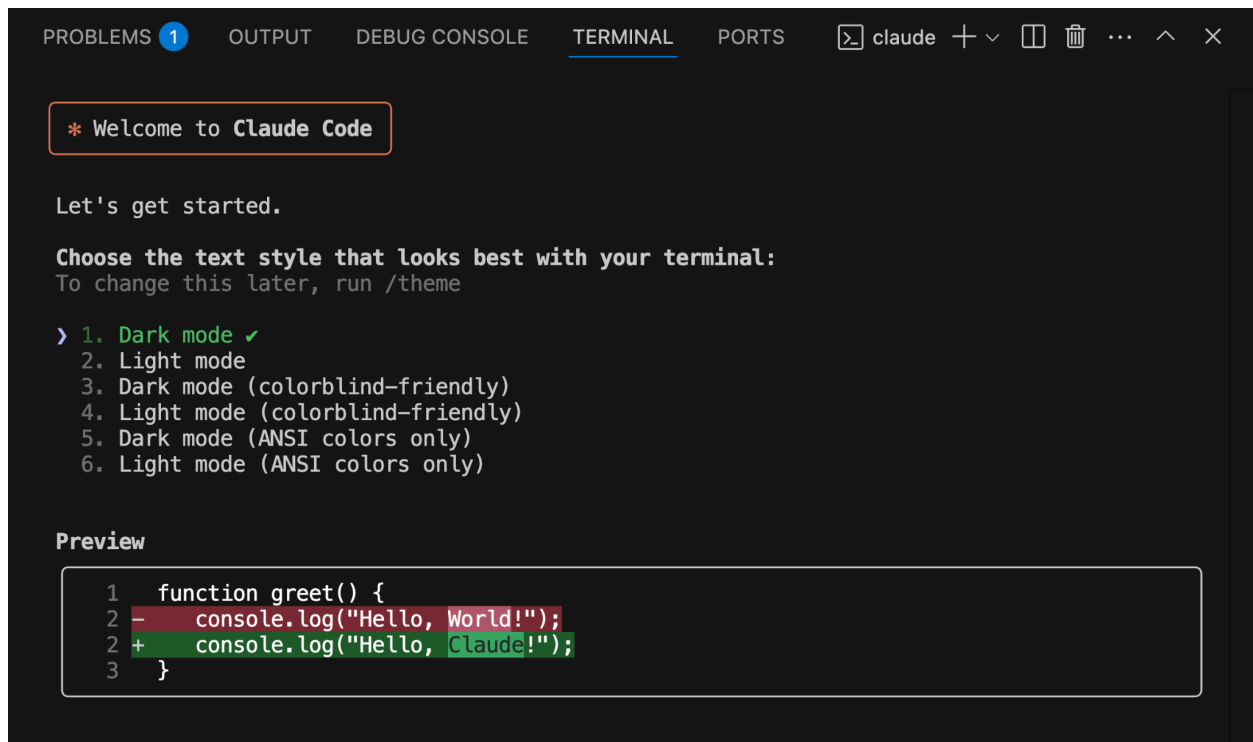
6. 배포된 devcontainer는 이미 `claude` 명령을 실행할 준비가 완료되어 있습니다. **claude** 명령을 바로 실행하고 클로드 구독에 대한 인증을 마치면, 호스트와 독립된 클로드 코드 실행 환경을 사용할 수 있습니다.

<VS Code Terminal박스>

```
$ claude
```

</VS Code Terminal박스>

[그림 4] VS Code 터미널에서 클로드 코드를 실행한 결과



```
PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS claude + - [ ] [X] ... ^ X

* Welcome to Claude Code

Let's get started.

Choose the text style that looks best with your terminal:
To change this later, run /theme

> 1. Dark mode ✓
  2. Light mode
  3. Dark mode (colorblind-friendly)
  4. Light mode (colorblind-friendly)
  5. Dark mode (ANSI colors only)
  6. Light mode (ANSI colors only)

Preview

1 function greet() {
2 - console.log("Hello, World!");
2 + console.log("Hello, Claude!");
3 }
```

<노트> 왜 devcontainer에서 클로드 코드를 실행하는지 모르겠어요. 이것 앞으로 사용하나요?

지금까지 실습을 진행했다면, 대체 이것 왜 진행하는거지 하는 의문이 들 수 있습니다.

이미 실습을 시작하기 전에 간단히 설명한 것처럼 클로드 코드와 같은 인공지능 에이전트를 사용하는 관점에서는 독립된 환경을 가지는 것이 보안적으로는 더 좋습니다. 사실 보안 수준은 높이면 높일수록 더 사용자에게는 번거로움을 유발합니다. 따라서 사용 목적 그리고 환경에 따라서 적절한 수준을 찾는 것이 중요합니다.

따라서 현재 작성한 devcontainer는 체험적인 목적으로 진행하고 책의 모든 진행은 로컬 호스트에서 하시는 것을 권장합니다. 현재 devcontainer는 아무 것도 구성되어 있지 않은 환경으로 실습 자체에 여러가지 번거로움을 유발할 수 있기 때문입니다.

</노트>

3. DevPod의 구성 변경

DevPod의 가장 큰 강점은 다양한 통합 개발 도구를 선택해서 사용할 수 있고, 도커 외에 다양한 제공자를 있는 것입니다. 따라서 VS Code 외에 Cursor를 사용해 보는 실습과 도커 외에 쿠버네티스를 사용하는 실습까지 진행해 DevPod가 가지는 뛰어난 점을 다시 한번 확인하도록 하겠습니다.

3.1 Cursor 지원

요즘 인기가 정말 많은 Cursor는 VS Code를 기반으로 만든 AI 내장 코드 에디터로, 기존 VS Code의 모든 기능과 확장 프로그램을 그대로 사용할 수 있습니다. 가장 큰 차이점은 Tab 자동완성, AI 채팅, 코드 생성 등의 AI 기능이 기본 내장되어 있어서, VS Code에서 별도 AI 확장 프로그램을 설치할 필요가 없다는 점입니다.

1. 배포했던 devcontainer를 **devpod delete devcontainer** 명령으로 지웁니다. 현업에서는 개발자의 성향마다 다양한 개발 환경을 동시에 관리하는 경우도 있고, 단일 환경을 선호하는 경우도 함께 존재합니다. 이는 개발자의 성향과 조직에 성격에 따른 것입니다. 현재 삭제하고 진행하는 부분은 혼동의 소지가 있기 때문이며, 설명한 것처럼 현업에서는 개별 프로젝트마다 조직에 따라 다르게 사용합니다.

<Terminal박스>

```
$ depod delete devcontainer

16:35:58 info Deleting container...

16:35:58 info Deleting devcontainer...

16:35:59 done Successfully deleted workspace 'devcontainer'
```

</Terminal박스>

2. 지워진 것을 확인하기 위해 **devpod list**를 실행해 다음과 같이 아무 것도 없는 상태임을 확인합니다.

<Terminal박스>

```
$ depod list

NAME | SOURCE | MACHINE | PROVIDER | IDE | LAST USED | AGE | PRO
-----+-----+-----+-----+-----+-----+-----+-----
```

</Terminal박스>

3. Cursor를 IDE로 사용하기 위해 **--ide cursor** 옵션을 주고, 그에 맞는 json을 호출하기 위해 **--devcontainer-path .devcontainer/devcontainer-cursor.json** 옵션도 함께 주고 실행합니다. 해당

값이 없을 경우 기본 값은 VS Code 그리고 .devcontainer/devcontainer.json 입니다. Cursor는 사전에 구성되어 있어야 하며, 배포가 완료되고 나면 자동으로 Cursor를 호출합니다.

<Terminal박스>

```
$ devpod up . --ide cursor --devcontainer-path .devcontainer/devcontainer-cursor.json
```

```
16:39:21 info Creating devcontainer...
```

```
16:39:25 info Build with docker buildx...
```

```
16:39:25 info #0 building with "container" instance using docker-container driver
```

```
16:39:25 info
```

```
16:39:25 info #1 [internal] load build definition from Dockerfile
```

```
16:39:25 info #1 transferring dockerfile: 2.59kB done
```

```
16:39:25 info #1 DONE 0.0s
```

[중략]

```
16:39:58 info Host network detected as: 172.17.0.0/24
```

```
16:39:58 info Firewall configuration complete
```

```
16:39:58 info Verifying firewall rules...
```

```
16:40:02 info Firewall verification passed - unable to reach https://example.com as expected
```

```
16:40:02 info Firewall verification passed - able to reach https://api.github.com as expected
```

```
16:40:02 done Successfully ran command : sudo /usr/local/bin/init-firewall.sh
```

```
16:40:02 info Install extensions
```

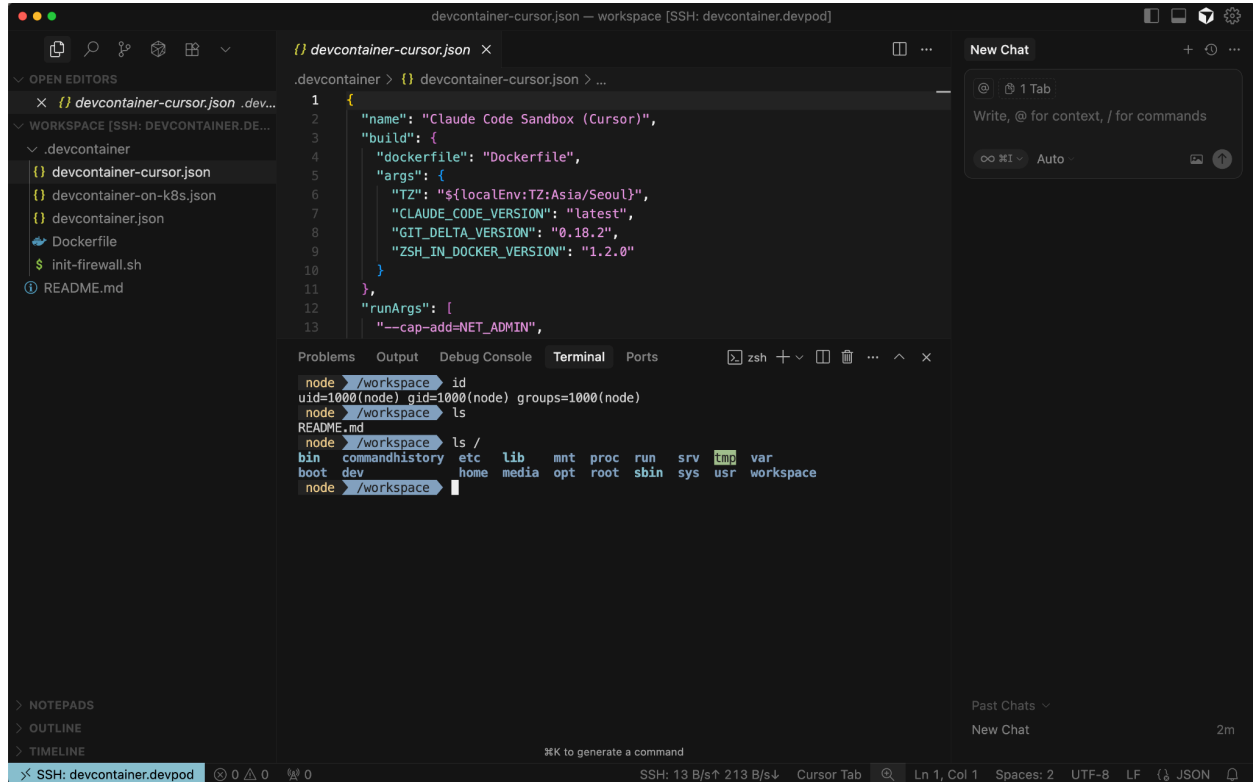
```
'anthropic.claude-code,dbaeumer.vscode-eslint,esbenp.prettier-vscode,eamodio.gitlens' in the background
```

```
16:40:02 info Run 'ssh devcontainer.devpod' to ssh into the devcontainer
```

```
16:40:02 info Starting Cursor...
```

</Terminal박스>

[그림 5] devpod로 생성된 devcontainer가 자동으로 Cursor 터미널에 연결됨



VS Code 외에 다른 인기 IDE인 Cursor 호출도 확인했으니, 이번에는 제공자를 도커에서 쿠버네티스로 변경해 보겠습니다.

3.2 쿠버네티스 지원

DevPod의 Kubernetes provider는 로컬 머신의 CPU/메모리 한계를 극복하기 위해 클러스터의 분산 리소스를 활용하여 개발 환경을 실행합니다. Docker provider는 로컬 머신에서 직접 컨테이너를 실행하므로 간단하고 빠르지만 하드웨어 제약이 있는 반면, Kubernetes provider는 설정이 복잡하지만 확장 가능한 리소스와 팀 간 일관된 환경을 제공합니다. 특히 대용량 프로젝트나 리소스 집약적인 개발 작업에서 Kubernetes provider가 로컬 리소스 부족 문제를 해결하는 효과적인 대안이 됩니다.

1. 역시 혼동이 올 수 있으니, **devpod delete devcontainer** 명령으로 진행했던 devcontainer를 지웁니다.

<Terminal>박스>

```
$ depod delete devcontainer
```

```
16:46:12 info Deleting container...
16:46:12 info Deleting devcontainer...
16:46:12 done Successfully deleted workspace 'devcontainer'
```

</Terminal박스>

2. 도커를 제공자로 추가했던 것처럼 쿠버네티스 또한 제공자로 추가해줘야 합니다.

<Terminal박스>

```
$ devpod provider add kubernetes
16:49:15 done Successfully installed provider kubernetes
16:49:15 done Successfully configured provider 'kubernetes'
```

</Terminal박스>

3. 기본 제공자가 아닌 쿠버네티스를 제공자로 사용할 것이므로 **--provider kubernetes**를 옵션으로 입력합니다. 그리고 **--devcontainer-path .devcontainer/devcontainer-on-k8s.json** 옵션으로 사용하여 배포되는 쿠버네티스 컨텍스트에 맞게 적용합니다. 해당 json은 AWS의 EKS에 방화벽 설정을 적용할 수 없어 해당 설정이 빠져 있습니다. 작업 전에 충분한 권한을 가지고 접근 가능한 쿠버네티스가 사전 구성되어 있어야 합니다.

<Terminal박스>

```
$ devpod up . --provider kubernetes --devcontainer-path .devcontainer/devcontainer-on-k8s.json
16:50:11 info Creating devcontainer...
16:50:16 info Create Persistent Volume Claim 'devpod-default-de-ee380'
16:50:16 info Create Pod 'devpod-default-de-ee380'
16:50:16 info Waiting for DevContainer Pod 'devpod-default-de-ee380' to come up...
16:50:21 info Waiting, since pod 'devpod-default-de-ee380' init container 'devpod-init' is waiting to start: (PodInitializing)
16:50:26 info Waiting, since pod 'devpod-default-de-ee380' init container 'devpod-init' is waiting to start: (PodInitializing)
16:50:31 info Download DevPod Agent...
16:50:42 info Setup container...
```


[중략]

16:51:42 info time="2025-09-27T07:51:42Z" level=info msg="Cmd: /bin/sh"

16:51:42 info time="2025-09-27T07:51:42Z" level=info msg="Args: [-c chmod +x /usr/local/bin/init-firewall.sh && echo \"node ALL=(root) NOPASSWD: /usr/local/bin/init-firewall.sh\" > /etc/sudoers.d/node-firewall && chmod 0440 /etc/sudoers.d/node-firewall]"

16:51:43 info time="2025-09-27T07:51:42Z" level=info msg="Util.Lookup returned: &{Uid:0 Gid:0 Username:root Name:root HomeDir:/root}"

16:51:43 info time="2025-09-27T07:51:42Z" level=info msg="Performing slow lookup of group ids for root"

16:51:43 info time="2025-09-27T07:51:42Z" level=info msg="Running: [/bin/sh -c chmod +x /usr/local/bin/init-firewall.sh && echo \"node ALL=(root) NOPASSWD: /usr/local/bin/init-firewall.sh\" > /etc/sudoers.d/node-firewall && chmod 0440 /etc/sudoers.d/node-firewall]"

16:51:43 info time="2025-09-27T07:51:43Z" level=info msg="USER node"

16:51:43 info time="2025-09-27T07:51:43Z" level=info msg="Cmd: USER"

16:51:43 info time="2025-09-27T07:51:43Z" level=info msg="Taking snapshot of full filesystem..."

16:51:51 info Chown projects...

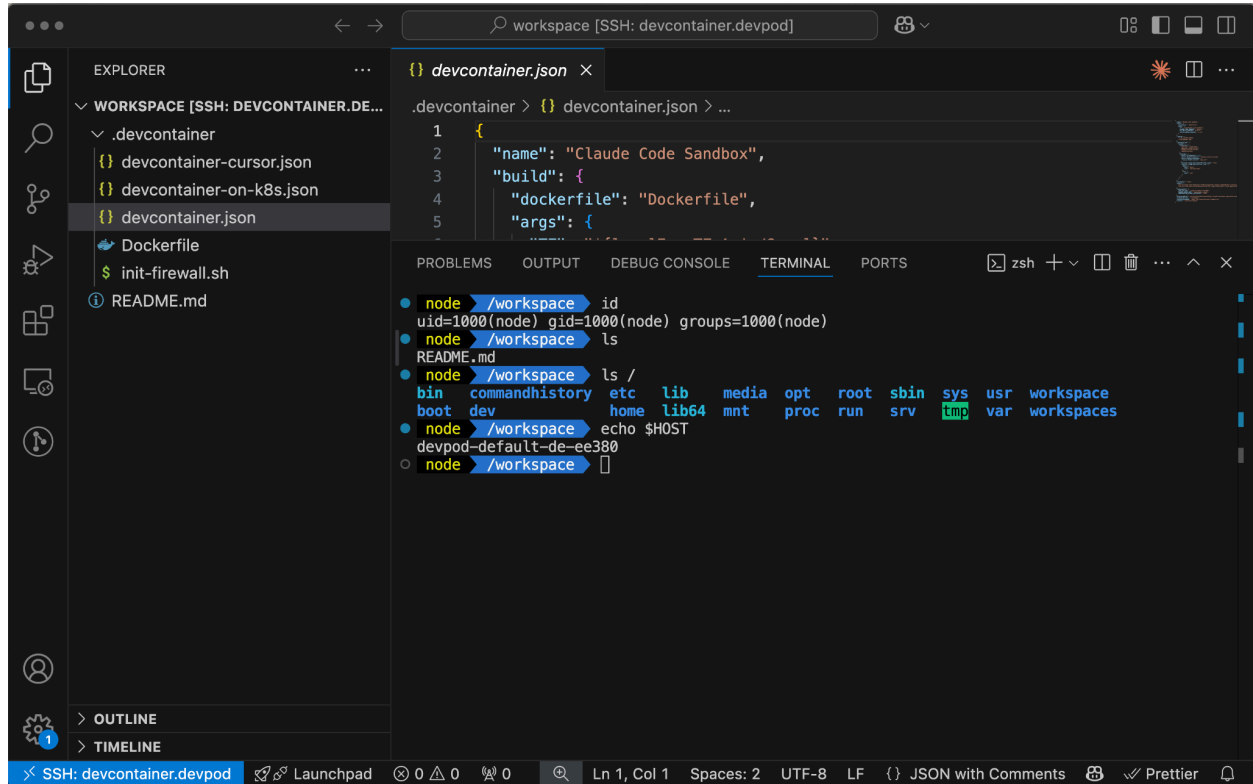
16:51:51 info Install extensions 'anthropic.claude-code,dbaeumer.vscode-eslint,esbenp.prettier-vscode,eamodio.gitlens' in the background

16:51:51 info Run 'ssh devcontainer.devpod' to ssh into the devcontainer

16:51:51 info Starting VSCode...

</Terminal박스>

[그림 6] devpod로 원격지에 있는 쿠버네티스에 devcontainer(devpod-default-de-ee380)가 생성됨



지금까지 devcontainer를 구성하기 위한 도구로 DevPod를 살펴보았고, 이를 통해 독립된 환경이 구성됨을 확인하였습니다. 주요 목적은 표준 개발 환경을 구성하는 것이나, 현재 문서는 클로드 코드를 위한 추가적인 목적이므로 표준 개발 환경을 구성하는 부분은 제외하였습니다.

<노트> DevPod와 연결되는 제공자와 지원되는 IDE

참고로 지원되는 다른 서비스 제공자와 IDE를 알면 좋을 것 같아서 다음과 같이 정리합니다.

<Terminal박스> 지원되는 서비스 제공자 (현재 구성된 것은 제외됨)

```
$ devpod provider list-available
```

```
List of available providers from loft:
```

```
aws
```

```
azure
```

</Terminal박스>

<Terminal박스> 지원 가능한 IDE (VS Code와 Cursor 외에 매우 많음)

```
$ devpod ide list
```

NAME	DEFAULT
-----+-----	
clion	false
codium	false
cursor	false
dataspell	false
fleet	false
goland	false
intellij	false
jupyternotebook	false
none	false
openvscode	false
phpstorm	false
positron	false
pycharm	false
rider	false
rstudio	false
rubymine	false
rustrover	false
vscode	false
vscode-insiders	false
webstorm	false
zed	false

</Terminal박스>